Open | SpeedShop™

How to Analyze the Performance of Parallel Codes 101

A Case Study with Open|SpeedShop

Half Day Tutorial @ SciDAC 2010
Chattanooga, TN

Why this tutorial?

- **Performance Analysis is becoming more and more important**
  - Complex architectures
  - Complex applications
  - Mapping applications onto architectures
- **Often hard to know where to start**
  - Which experiments to run first?
  - How to plan follow on experiments?
  - What kind of problems can be explored?
  - How to interpret the data?

Tutorial Goals

- **Provide basic guidance on …**
  - How to understand the performance of a code?
  - How to answer basic performance questions?
  - How to plan experiments?
- **Basics on Open|SpeedShop**
  - Introduction into one possible solution
  - Basic usage instructions
  - Pointers to additional documentation
- **Provide you with the ability to …**
  - Run these experiments on your own code
  - Provide starting point for performance optimizations

Why Open|SpeedShop?

- **Open Source Performance Analysis Tool Framework**
  - Most common performance analysis steps ***all in one tool***
  - ***Extensible*** by using plugins for data collection and representation
- **Flexible and Easy to use**
  - User access through ***GUI***, ***Command Line***, and ***Python Scripting***
- **Several Instrumentation Options**
  - All work on ***unmodified application binaries***
  - ***Offline*** and ***online data collection*** / ***attach*** to running applications
- **Target: Cluster systems and MPPs**
  - ***Linux Clusters*** with x86, IA-64, Opteron, and EM64T CPUs
  - ADD BG/P & XT PORT
- **Status & Availability**
  - Version 1.9.3.4 about to be released / working on large lab codes
  - Distribution and CVS access available through sourceforge.net

“Rules”

- **Let’s keep this interactive**
  - Feel free to ask as we go along
  - Online demos as we go along
- **Feedback on O|SS**
  - What is good/missing in the tool?
  - What should be done differently?
  - Please report bugs/incompatibilities

Presenters

- **Martin Schulz, LLNL**
- **Don Maghrak, Krell**

Larger Team:

- Jim Galarowicz, Krell
- David Montoya, LANL
- Mahesh Rajan, Sandia
- William Hachfeld & Dave Whitney, Krell
- Samuel Gutierrez & Dane Gardner, LANL
- Scott Cranford & Joseph Kenny, Sandia NLs
- Chris Chambreau, LLNL

Outline

- Concepts in performance analysis
- Introduction into Open|SpeedShop
- How to understand profiles?
- How to relate data to architectural properties?
- How to find I/O bottlenecks?
- How to find bottlenecks in parallel codes?
- How can I repeat this at home?
  What else can I do with O|SS?

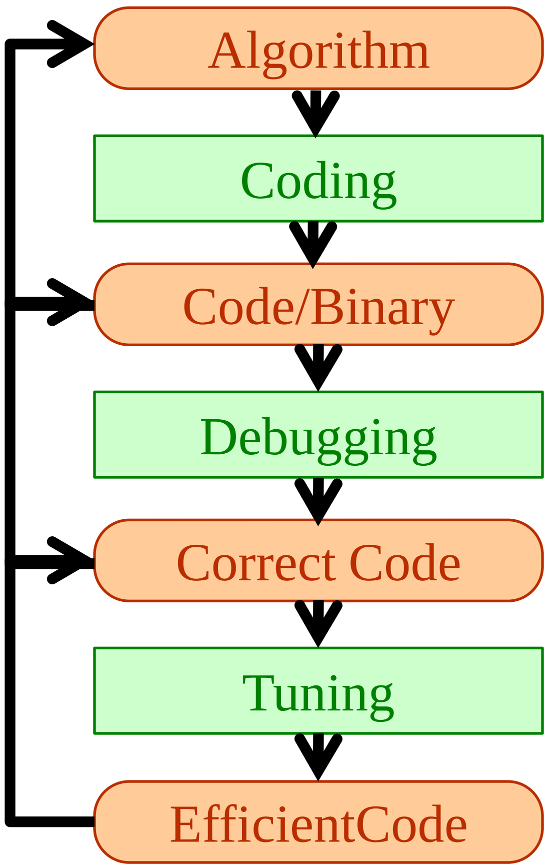
Open | SpeedShop™

Section 1
Concepts in Performance Analysis

How to Analyze the Performance of Parallel Codes 101
A Case Study with Open|SpeedShop

Development Cycle

- **Performance Tuning is an essential part of development**
  - Part of development cycle
  - Potential impact on every stage
  - Should be done from initial stages of code development
- **Typical use**
  - Measure performance
  - Analyze data
  - Modify code and/or algorithm
  - Repeat measurements
  - Analyze differences



Available Support

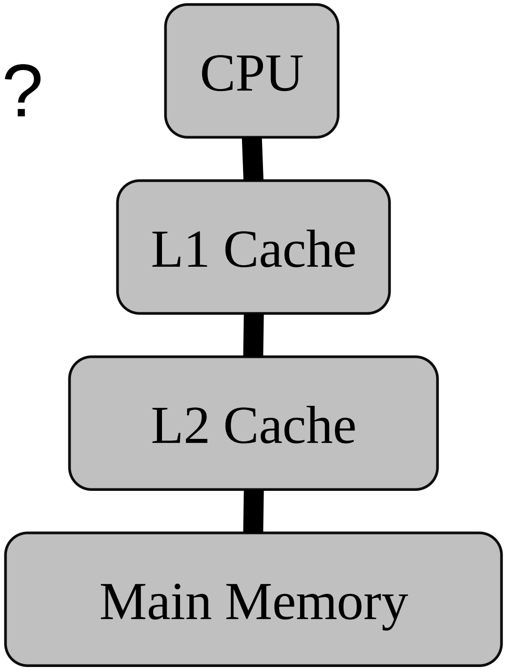
- **First line of defense**
  - Full execution timings
  - Comparison between input parameters
  - Historical trends
- **Disadvantage**
  - Coarse grain measurements
  - Can’t pin performance bottlenecks
  - Alternative: code integration
    - Hard to maintain
    - Requires in-depth, a-priori code knowledge
- **Need for performance analysis tools**

What Can Tools Do For You?

- **Gather fine grain performance data**
  - Low intrusive instrumentation
  - Adaptive granularity
- **Relation to source code**
  - Connect performance to source lines
  - Enable root cause analysis
- **But: usage is often a “black art”**
  - Many options and usage scenarios
  - Interpretation of results often not intuitive

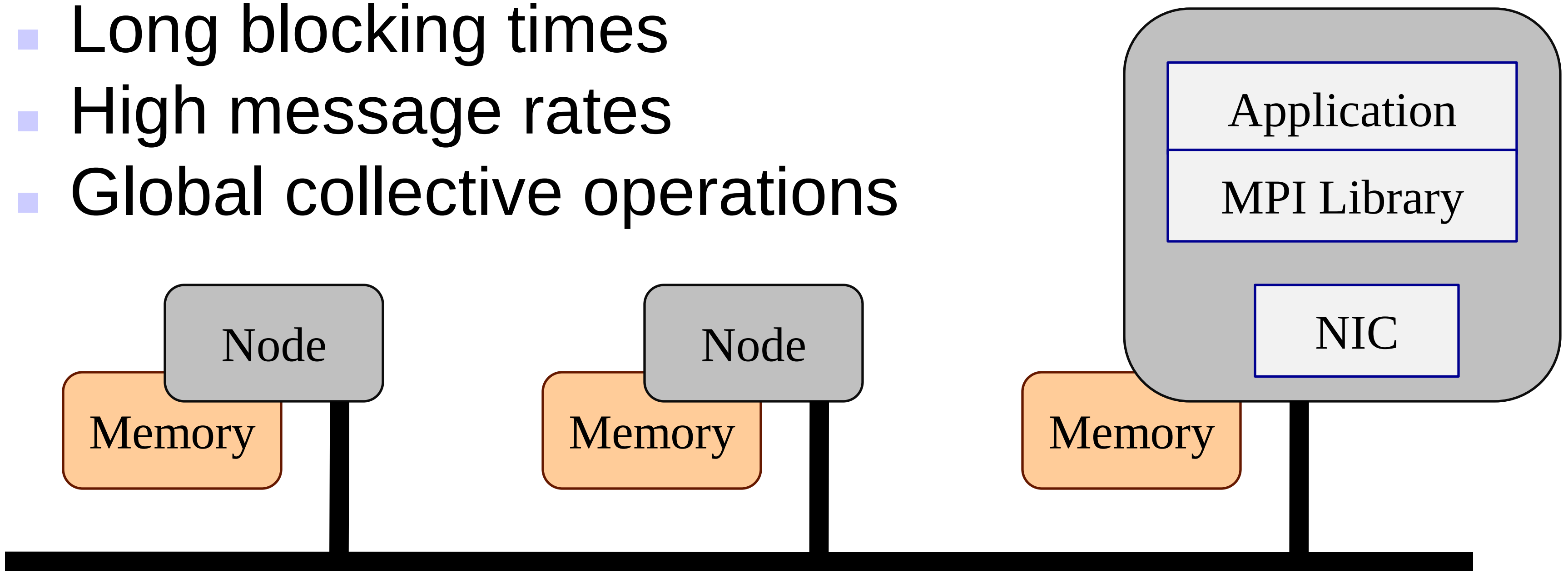
Questions: Sequential runs

- **Identify computational parts**
  - Where am I spending my time?
  - Does this match intuition / computational kernels?
- **Impact of cache hierarchies**
  - Do I have excessive cache misses?
  - How is my data locality?
  - Impact of TLB misses?
- **External resources**
  - Is my I/O efficient?
  - Shared libraries?



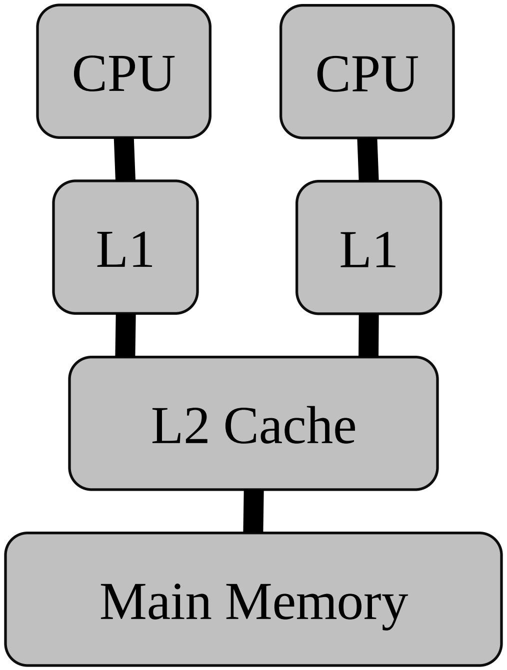
Questions: MPI Codes

- **Distributed memory model**
  - Sequential optimizations for each task
  - Inter-process message optimizations
- **Issues to look for:**
  - Long blocking times
  - High message rates
  - Global collective operations



Questions: Threaded Codes

- **Shared memory model**
  - Single shared storage accessible from all CPUs
  - Most common models: POSIX threads, OpenMP
  - Locality optimizations apply
- **Issues to look for:**
  - Shared memory buses/bandwidth
  - Synchronization overhead
  - Thread startup
  - Sufficient work per thread?
- **Complications:**
  - NUMA architectures
    - Memory allocation policy?
  - Architectural differences



Two Types of Tools

- **Sampling Experiments**
  - Periodically interrupt run and record location
  - Report statistical distribution of these locations
  - Data aggregated over time
  - Typically provides good overview
  - Overhead mostly low and uniform
- **Tracing Experiments**
  - Gather and store individual application events, e.g., function invocations (MPI, I/O, …)
  - Keep timing information
  - Provides detailed, low-level information
  - Higher overhead, potentially bursty

Analysis Frameworks

- **All analysis options in one "box"**
  - Different experiments
  - Combine tracing and sampling
  - Integrated analysis options
  - Uniform access to storing data
- **Advantage:**
  - Integrate into workflow only once
  - Lower learning curve
  - Reuse source annotations
  - Easier installation and maintenance

Instrumentation Options

- **Instrumentation**
  - How to add data acquisition into codes?
  - Integral part of any tools
- **Binary methods**
  - Library preloading/function interception
  - Static binary rewriting
  - Dynamic binary instrumentation
- **Source code methods**
  - Explicit user annotations
  - Source-to-source transformation
  - Compiler flags

Existing Tools

- **Basic OS tools**
  - time, gprof
- **Hardware counters**
  - PAPI APIs & tool set
  - hwctime (AIX)
- **TAU (U. of Oregon)**
  - Comprehensive tool set
  - Automatic source code instrumentation
  - 3D visualization
- **HPC Toolkit (Rice)**
  - Binary preloading
  - Sampled measurements
- **Scalasca (Juelich)**
  - Profiling
  - Automatic trace analysis
- **Vendor tools**
  - Cray Pat
  - Vtune (Intel)
  - HPCToolkit (IBM)
- **Specialize tools**
  - Paradyn (U. of Wisc.)
    Adaptive instrumentation
  - Libra (LLNL)
    Load balance analysis
- **Open|SpeedShop**

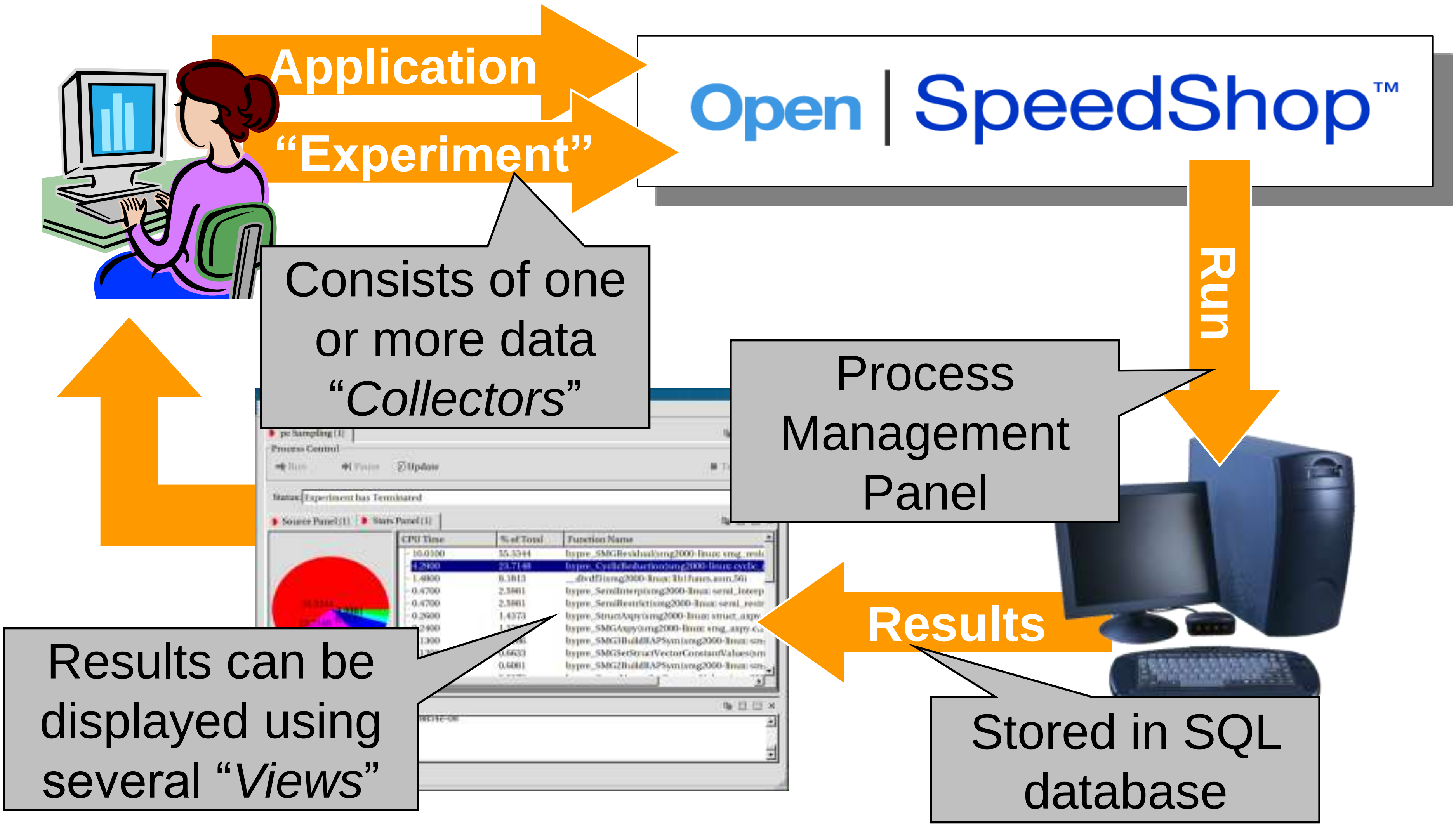
How to Pick a Tool

- **Define the questions you want answered**
  - Overview vs. detailed measurements?
  - What part of the system to look at?
  - Needed: suspicion about ill-performing codes
- **Instrumentation approach**
  - Convenience wrt. workflow
  - Affordable overhead
- **Frameworks**
  - Experiments vs. new tool
  - Extent of planned performance analysis
  - Generally lower learning curve overall

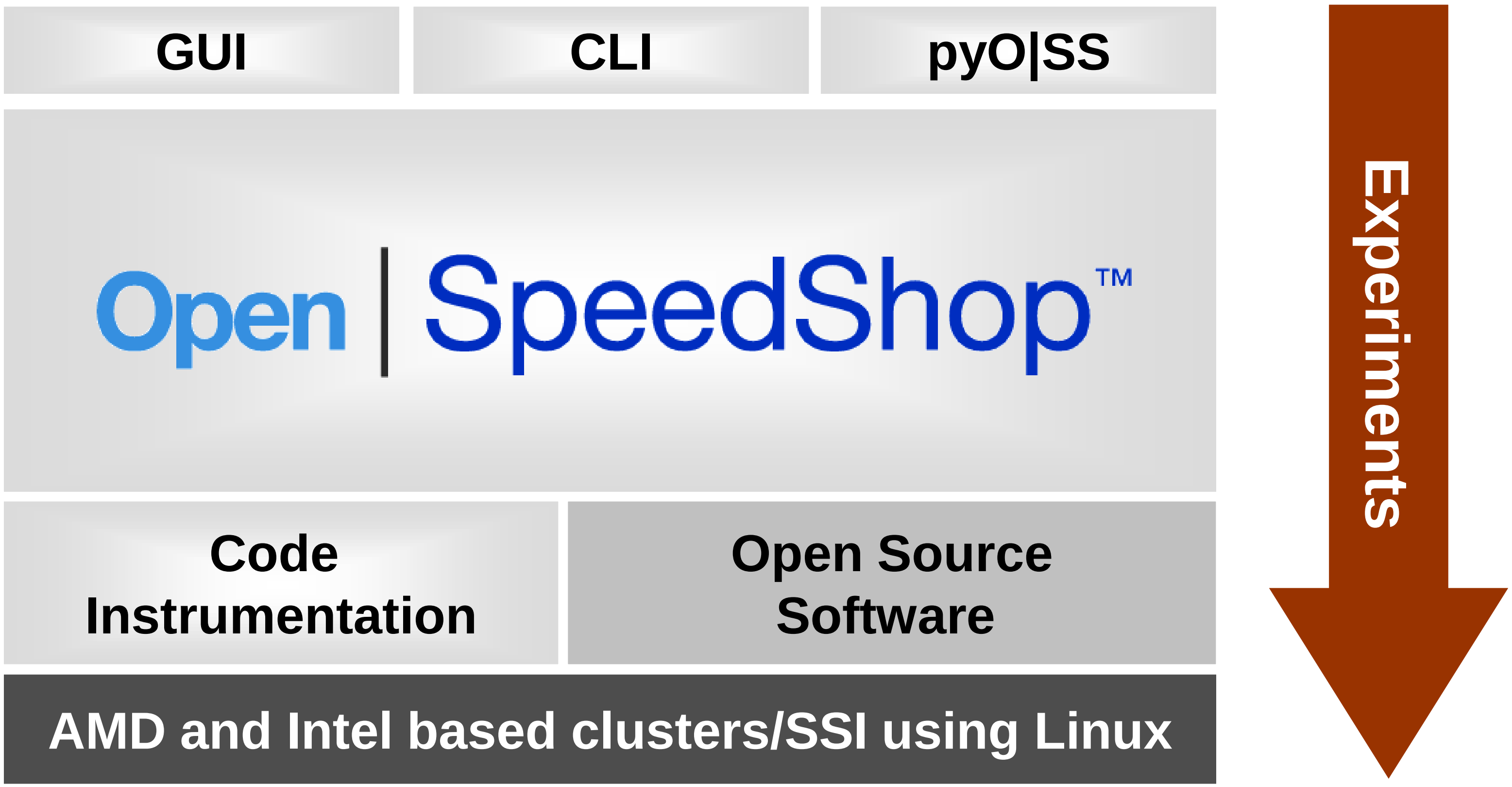
Section 2
Introduction into Open|SpeedShop

How to Analyze the Performance of Parallel Codes 101
A Case Study with Open|SpeedShop

Experiment Workflow

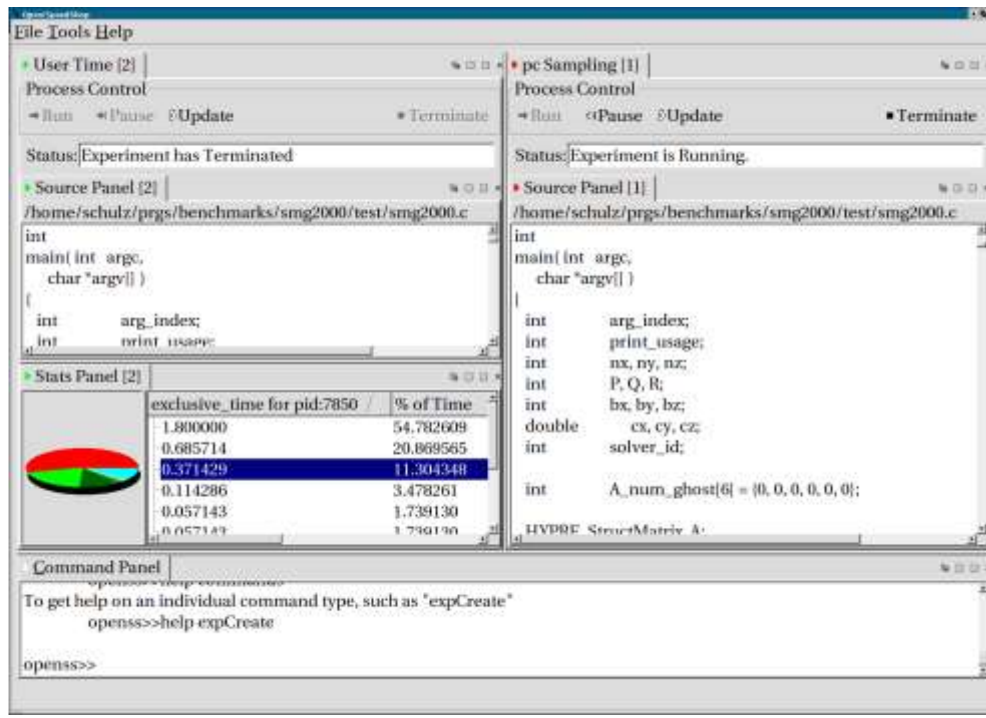


High-level Architecture



Basic Interface

- **Step 1:**
  - Gather data from command line
  - Example: osspcsamp “<application>”
  - Create database
- **Step 2:**
  - Analyze data in GUI
  - Simple graphics
  - Relation to source code



Advanced Interfaces

- **Scripting language**
  - Batch interface
  - O|SS command line (CLI)
- **Python module**

```
Experiment Commands
    expAttach
    expCreate
    expDetach
    expGo
    expView

List Commands
    list –v exp
```

```
import openss

my_filename=openss.FileList("myprog.a.out")
my_exptype=openss.ExpTypeList("pcsamp")
my_id=openss.expCreate(my_filename,my_exptype)

openss.expGo()

My_metric_list = openss.MetricList("exclusive")
my_viewtype = openss.ViewTypeList("pcsamp")
result = openss.expView(my_id,my_viewtype,my_metric_list)
```

Performance Experiments

- **Concept of an Experiment**
  - What to measure and analyze?
  - Experiment chosen by user
  - Any experiment can be applied to any code
- **Consists of Collectors and Views**
  - Collectors define specific data sources
    - Hardware counters
    - Tracing of library routines
  - Views specify data aggregation and presentation
  - Multiple collectors per experiment possible

Sampling Experiments

- **PC Sampling (pcsamp)**
  - Record PC in user defined time intervals
  - Low overhead overview of time distribution
- **User Time (usertime)**
  - PC Sampling + Call stacks for each sample
  - Provides inclusive & exclusive timing data
- **Hardware Counters (hwc, hwctime)**
  - Sample HWC overflow events
  - Access to data like cache and TLB misses
  - Default event is PAPI_TOT_CYC overflows

Tracing Experiments

- **I/O Tracing (io, iot)**
  - Record invocation of all POSIX I/O events
  - Provides aggregate and individual timings
- **MPI Tracing (mpi, mpit, mpiotf)**
  - Record invocation of all MPI routines
  - Provides aggregate and individual timings
- **Floating Point Exception Tracing (fpe)**
  - Triggered by any FPE caused by the code
  - Helps pinpoint numerical problem areas

Parallel Experiments

- **O|SS supports MPI and threaded codes**
  - Tested with a variety of MPI implementation
  - Thread support based on POSIX threads
  - OpenMP supported through POSIX threads
- **Any experiment can be parallel**
  - Automatically applied to all tasks/threads
  - Default views aggregate across all tasks/threads
  - Data from individual tasks/threads available
- **Specific parallel experiments (e.g., MPI)**

Running a First Experiment

1. **Picking the experiment**
   - What do I want to measure?
   - We will start with pcsamp to get a first overview
2. **Launching the application**
   - How do I control my application under O|SS?
   - osspcsamp “mpirun –np 2 smg2000 –n 80 80 80”
3. **Storing the results**
   - O|SS will create a database
   - Name: smg2000-pcsamp.openss
4. **Exploring the gathered data**
   - O|SS will print default output
   - Open GUI to analyze data in detail (run: openss)

Example Run with Output

osspcsamp "./smg2000 –n 80 80 80"

```
[jeg@localhost test]$ osspcsamp "./smg2000 -n 80 80 80"
[openss]: pcsamp experiment using the pcsamp experiment default sampling rate: "100".
[openss]: Using OPENSS_PREFIX installed in /opt/OSS-mrnet
[openss]: Setting up offline raw data directory in /tmp/jeg/offline-oss
[openss]: Running offline pcsamp experiment using the command:
"./smg2000 -n 80 80 80"

Running with these driver parameters:
  (nx, ny, nz)    = (80, 80, 80)
  (Px, Py, Pz)    = (1, 1, 1)
  (bx, by, bz)    = (1, 1, 1)
  (cx, cy, cz)    = (1.000000, 1.000000, 1.000000)
  (n_pre, n_post) = (1, 1)
  dim             = 3
  solver ID       = 0
=============================================
Struct Interface:
=============================================
Struct Interface:
  wall clock time = 0.080000 seconds
  cpu clock time  = 0.060000 seconds
=============================================
Setup phase times:
=============================================
SMG Setup:
  wall clock time = 0.680000 seconds
  cpu clock time  = 0.660000 seconds
=============================================
Solve phase times:
=============================================
SMG Solve:
  wall clock time = 4.810000 seconds
  cpu clock time  = 4.800000 seconds

Iterations = 7
Final Relative Residual Norm = 2.844100e-07


[openss]: Converting raw data from /tmp/jeg/offline-oss into temp file X.0.openss

Processing raw data for smg2000
Processing processes and threads ...
Processing performance data ...
Processing functions and statements ...

[openss]: Restoring and displaying default view for:
        /home/jeg/DEMOS/demos/sequential/smg2000/test/smg2000-pcsamp.openss
[openss]: The restored experiment identifier is:  -x 1
                                                                46,0-1         Top
```

Example Run with Output

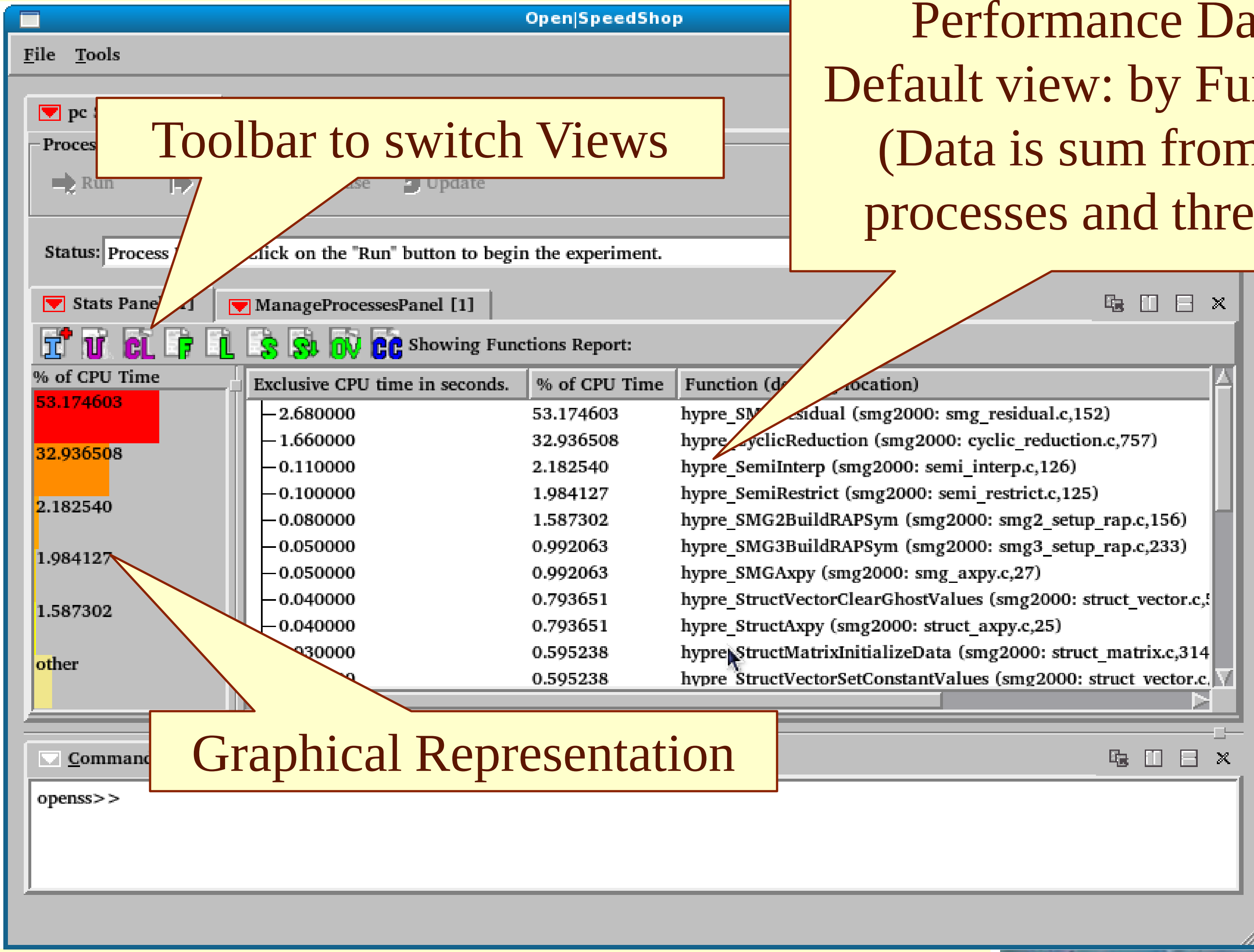
osspcsamp "./smg2000 –n 80 80 80"

jeg@localhost:~/DEMOS/demos/sequential/smg2000/test

File Edit View Terminal Help

```
Solve phase times:
=============================================
SMG Solve:
  wall clock time = 4.810000 seconds
  cpu clock time  = 4.800000 seconds

Iterations = 7
Final Relative Residual Norm = 2.844100e-07


[openss]: Converting raw data from /tmp/jeg/offline-oss into temp file X.0.openss

Processing raw data for smg2000
Processing processes and threads ...
Processing performance data ...
Processing functions and statements ...

[openss]: Restoring and displaying default view for:
        /home/jeg/DEMOS/demos/sequential/smg2000/test/smg2000-pcsamp.openss
[openss]: The restored experiment identifier is:  -x 1

  Exclusive CPU time         % of CPU Time  Function (defining location)
         in seconds.
            2.680000             53.174603  hypre_SMGResidual (smg2000: smg_residual.c,152)
            1.660000             32.936508  hypre_CyclicReduction (smg2000: cyclic_reduction.c,757)
            0.110000              2.182540  hypre_SemiInterp (smg2000: semi_interp.c,126)
            0.100000              1.984127  hypre_SemiRestrict (smg2000: semi_restrict.c,125)
            0.080000              1.587302  hypre_SMG2BuildRAPSym (smg2000: smg2_setup_rap.c,156)
            0.050000              0.992063  hypre_SMGAxpy (smg2000: smg_axpy.c,27)
            0.050000              0.992063  hypre_SMG3BuildRAPSym (smg2000: smg3_setup_rap.c,233)
            0.040000              0.793651  hypre_StructVectorClearGhostValues (smg2000: struct_vector.c,592)
            0.040000              0.793651  hypre_StructAxpy (smg2000: struct_axpy.c,25)
            0.030000              0.595238  hypre_StructMatrixInitializeData (smg2000: struct_matrix.c,314)
            0.030000              0.595238  hypre_StructVectorSetConstantValues (smg2000: struct_vector.c,537)
            0.020000              0.396825  hypre_CycRedSetupCoarseOp (smg2000: cyclic_reduction.c,211)
            0.020000              0.396825  hypre_StructInnerProd (smg2000: struct_innerprod.c,32)
            0.020000              0.396825  hypre_StructMatrixSetBoxValues (smg2000: struct_matrix.c,458)
            0.020000              0.396825  main (smg2000: smg2000.c,21)
            0.010000              0.198413  hypre_CreateCommInfoFromStencil (smg2000: communication_info.c,58)
            0.010000              0.198413  _int_malloc (libc-2.10.2.so: malloc.c,0)
            0.010000              0.198413  hypre_CommTypeDestroy (smg2000: communication.c,826)
            0.010000              0.198413  __brk (libc-2.10.2.so)
            0.010000              0.198413  hypre_StructVectorSetBoxValues (smg2000: struct_vector.c,258)
            0.010000              0.198413  hypre_BoxArrayArrayCreate (smg2000: box.c,106)
            0.010000              0.198413  hypre_SMGSetupInterpOp (smg2000: smg_setup_interp.c,88)
            0.010000              0.198413  hypre_SMGRelaxSetup (smg2000: smg_relax.c,357)
            0.010000              0.198413  hypre_SMGSetStructVectorConstantValues (smg2000: smg.c,379)


[jeg@localhost test]$
```

Default Data View

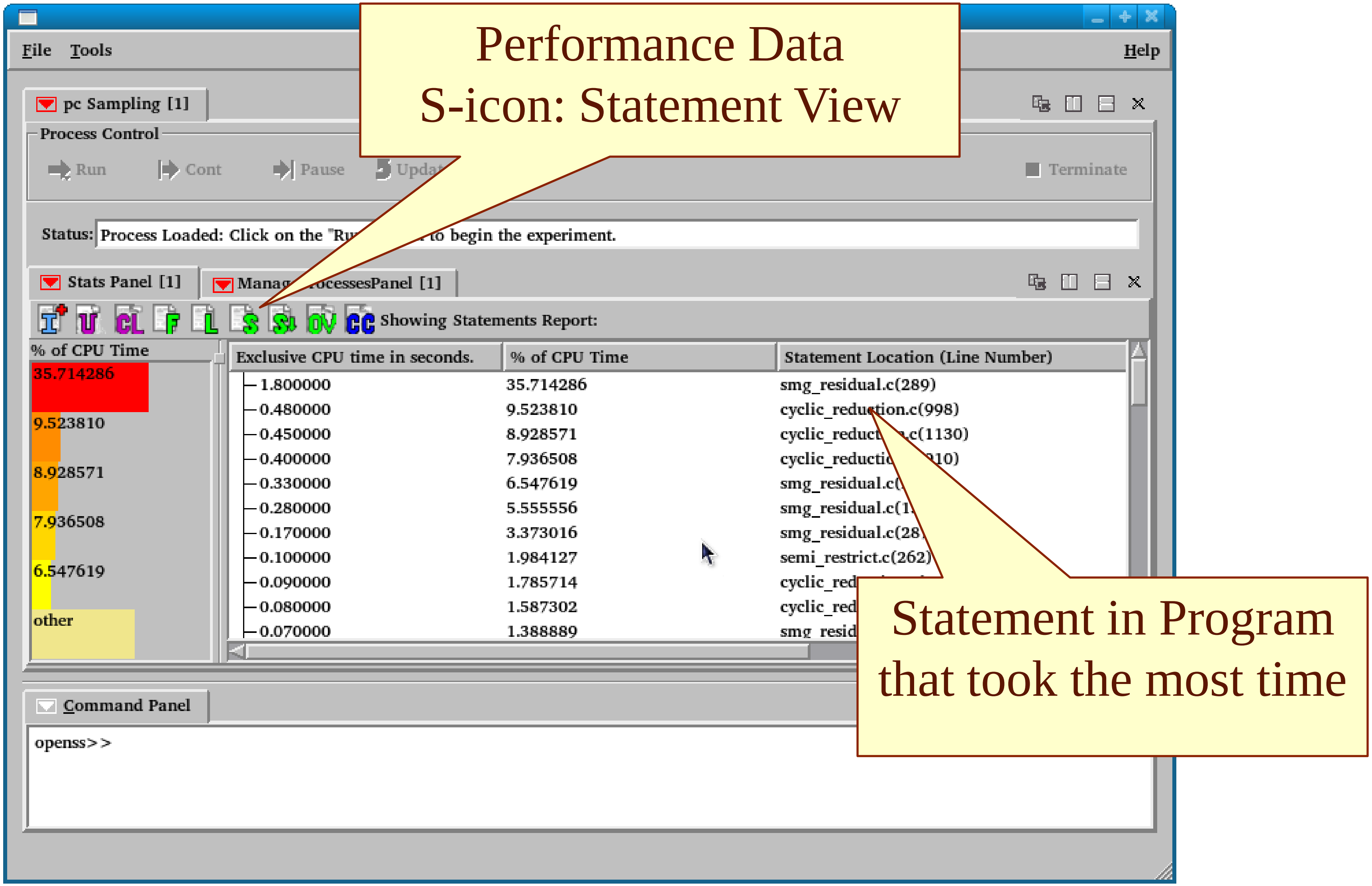


Toolbar to switch Views

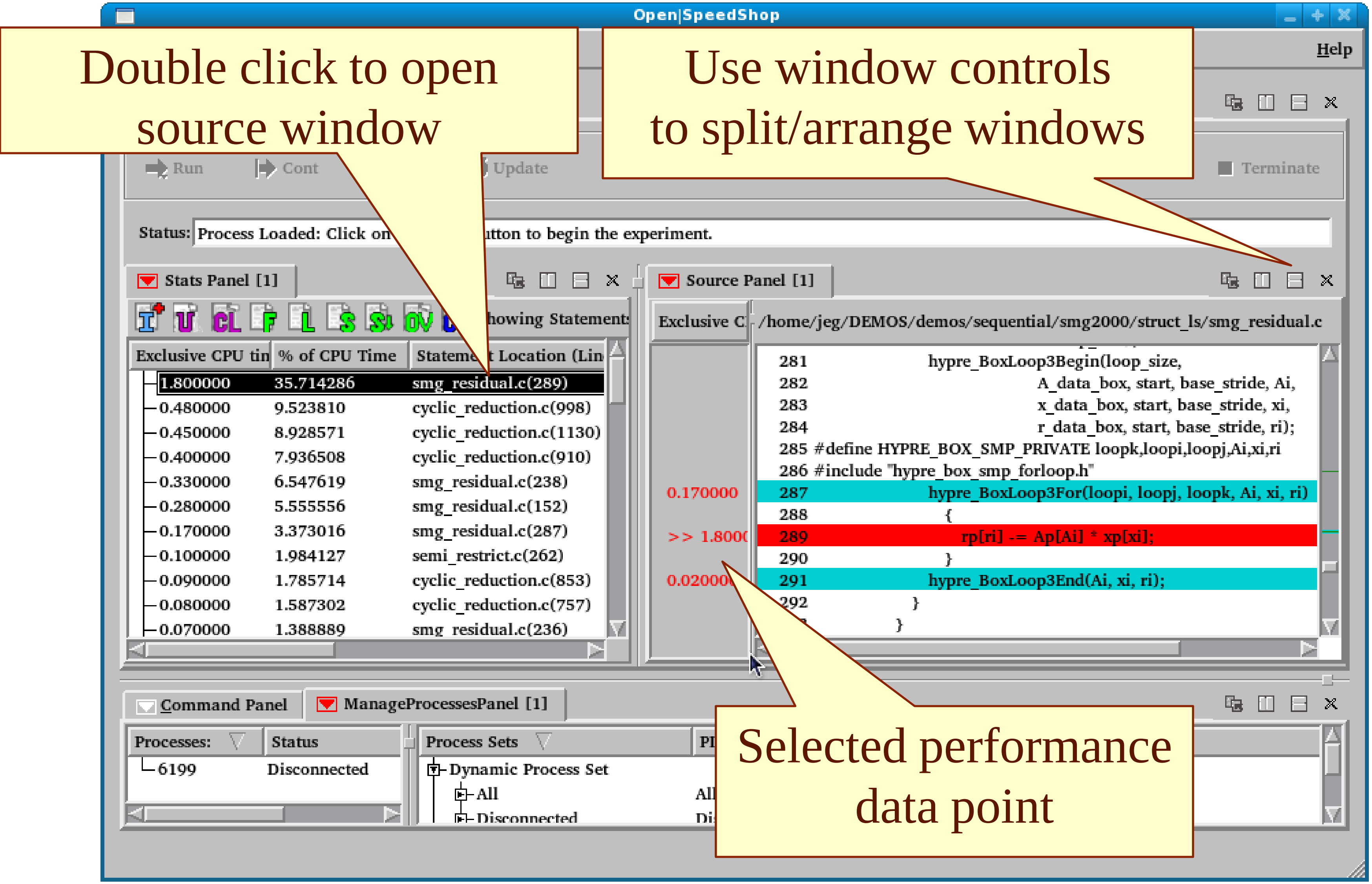
Performance Data
Default view: by Function
(Data is sum from all
processes and threads)

Graphical Representation

Statements Data View



Associate Source & Data



Summary

- **Open|SpeedShop is a comprehensive framework for performance analysis**
  - Binary instrumentation at runtime
  - Support for profiling and tracing
- **Predefined experiments**
  - Flat & Context profiles, hardware counters
  - MPI, I/O, FPE tracing
- **Multiple user interfaces**
  - GUI, Batch, Interactive, Python
  - Fully compatible

Open | SpeedShop™

Section 3
How to Understand Profiles?

How to Analyze the Performance of Parallel Codes 101
A Case Study with Open|SpeedShop

Using Profiles

- **What is a profile?**
  - Aggregate measurements (during collection)
  - Over time and code sections
- **Why use a profile?**
  - Reduced size of performance data
  - Typically collected with low overhead
  - Provides good overview of performance
- **Disadvantages**
  - Require a-priori definition of aggregation
  - Omits performance details of individual events
  - Possible sampling frequency skew

Standard Profiling Techniques

- **Statistical Performance Analysis**
  - Interrupt execution in periodic intervals
  - Record location of execution (PC value)
  - Optionally annotate with additional data
    - Stack traces
    - Hardware counters
  - Count equivalent samples
- **Advantages**
  - Low Overhead
  - Low Perturbation
  - Good to Get Overview / Find Hotspots

Sampling Experiments in O|SS

- **PC Sampling**
  - Approximates CPU Time For Line and Function
  - No Call Stacks
  - Script: osspcsamp
- **User Time**
  - Inclusive vs. Exclusive Time
  - Includes Call stacks
  - Script: ossusertime
- **HW Counters**
  - Samples Hardware Counter Overflows
  - Script: osshwc and osshwctime (with callstacks)

Step 1: Flat Profile

- **Answers a basic question:**
  - Where does my code spend its time?
- **Representation**
  - List of code elements
    - Varying granularity
    - Statements, Functions, …
  - Time spent at each function
- **Flat profiles through sampling**
  - Alternative to overhead of direct measurements
  - Add contributions taken from samples
  - Requires sufficient number of samples

Running Usertime

Offline pcsamp Experiment – smg2000

Option 1: Basic syntax

osspcsamp “smg2000 –n 50 50 50”

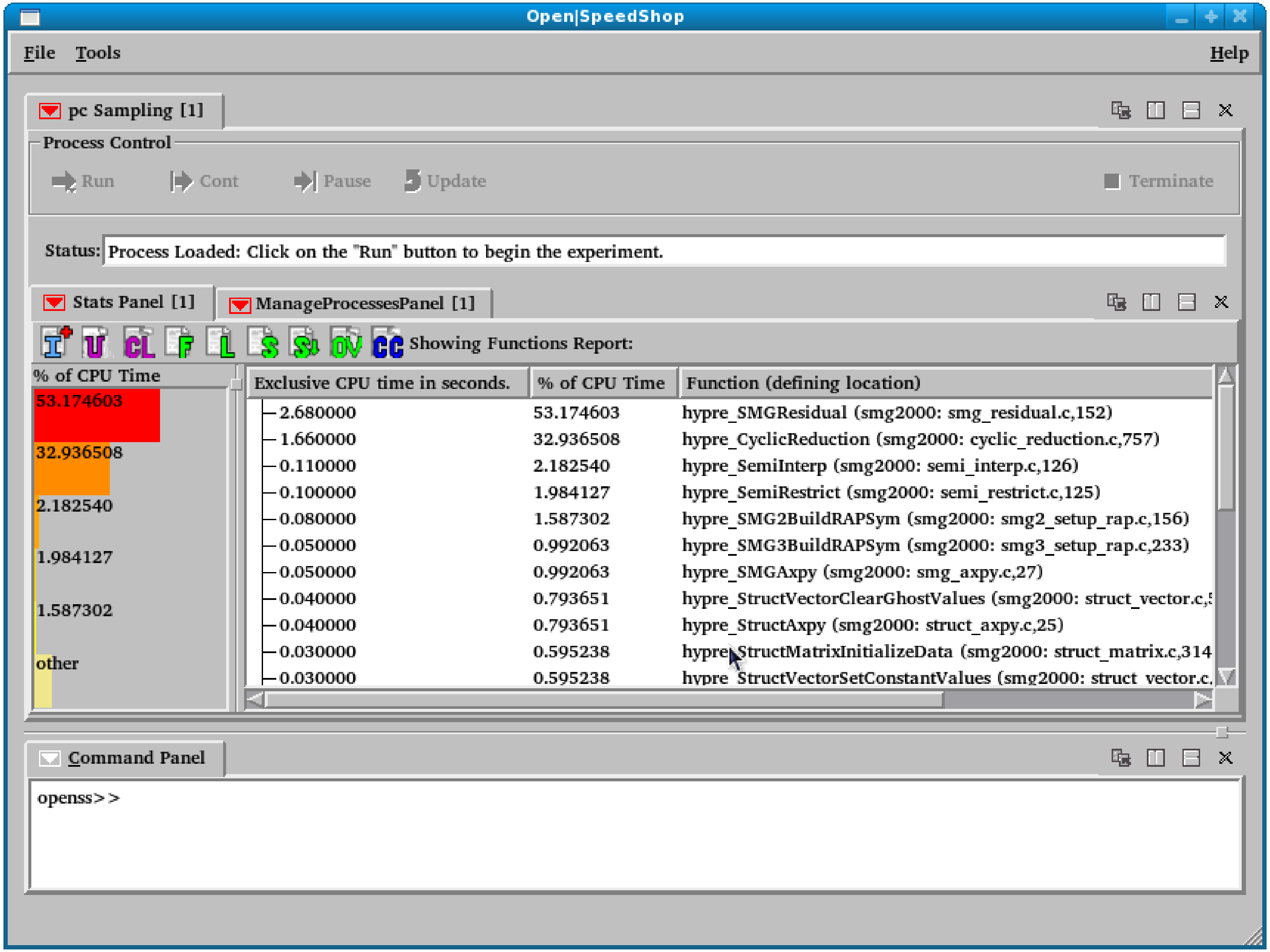
- **Parameters**
  - Sampling frequency (samples per second)
  - Additional parameter: high | low | default
    OR actual frequency (default is 100)

Option 2: Explicit Syntax

openss –offline –f “smg2000 –n 50 50 50” pcsamp

Recommendation: compile code with –g to get statements!

Viewing Flat Profiles



Identifying Critical Regions

- **Profiles show computationally intensive regions of the code**
  - First views: per functions or per statement
  - Questions:
    - Are those functions/statements expected?
    - Do they match the computational kernels?
    - Any runtime functions?
- **Identify bottleneck components**
  - Profile aggregated by shared objects
  - Correct/expected modules?
  - Impact of support & runtime libraries

Adding Context

- **Missing information in flat profiles**
  - Distinguish routines call from multiple callers
  - Understand invocation history
  - Context for performance data
- **Critical technique: Stack traces**
  - Gather stack trace for each sample
  - Aggregate only samples with equal trace
- **User perspective:**
  - Butterfly views (caller/callee relationships)
  - Hot call paths

The Usertime Experiment

- **Provides inclusive/exclusive time**
  - Time spent inside a function
  - Time spent in a function and its children
- **Similar to pcsamp experiment**
  - Collect pcsamp information
  - Collect call stack information at every sample
- **Tradeoffs**
  - Additional context information
  - Higher overhead/lower sampling rate

Running Usertime

Offline usertime Experiment – smg2000

<u>Option 1: Basic syntax</u>

ossusertime “smg2000 –n 50 50 50”

- **Parameters**
  - Sampling frequency (samples per second)
  - Additional parameter: high | low | default
    OR actual frequency (default is 35)

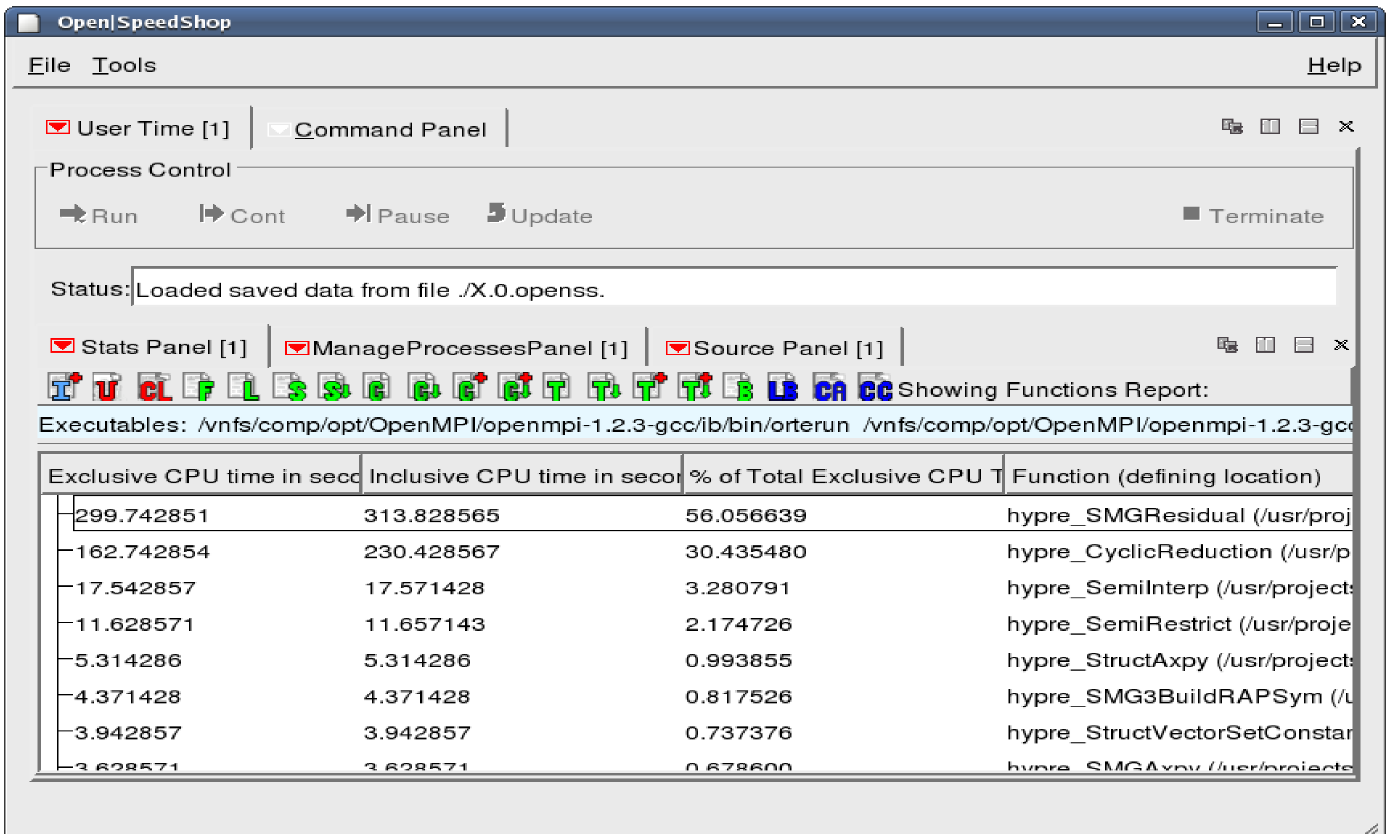
<u>Option 2: Explicit Syntax</u>

openss –offline –f “smg2000 –n 50 50 50” usertime

Recommendation: compile code with –g to get statements!

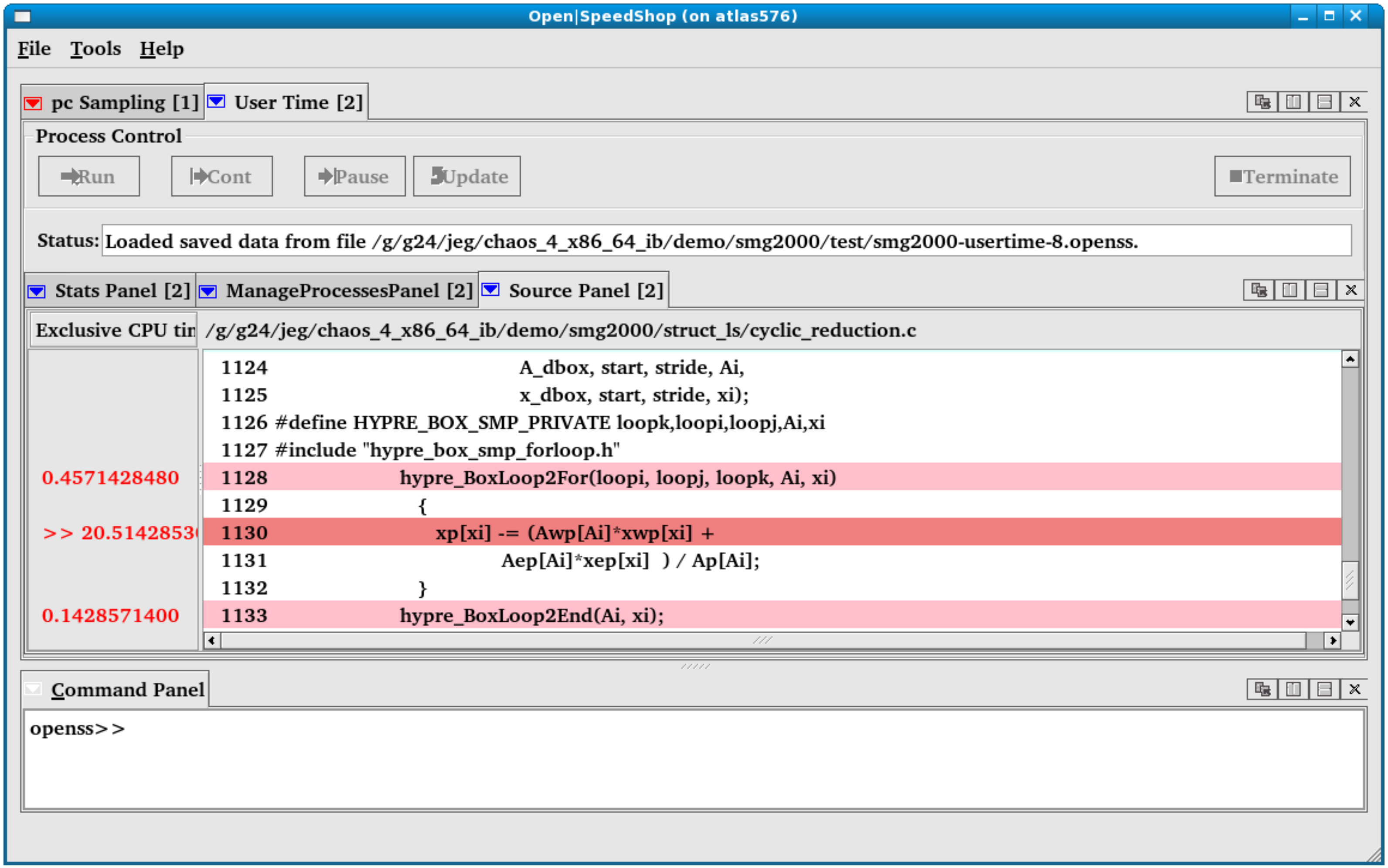
Viewing the Usertime Experiment

- **Default View**
  - Similar to pcsamp view from first example
  - Calculates inclusive vs. exclusive times



Source Code Mapping

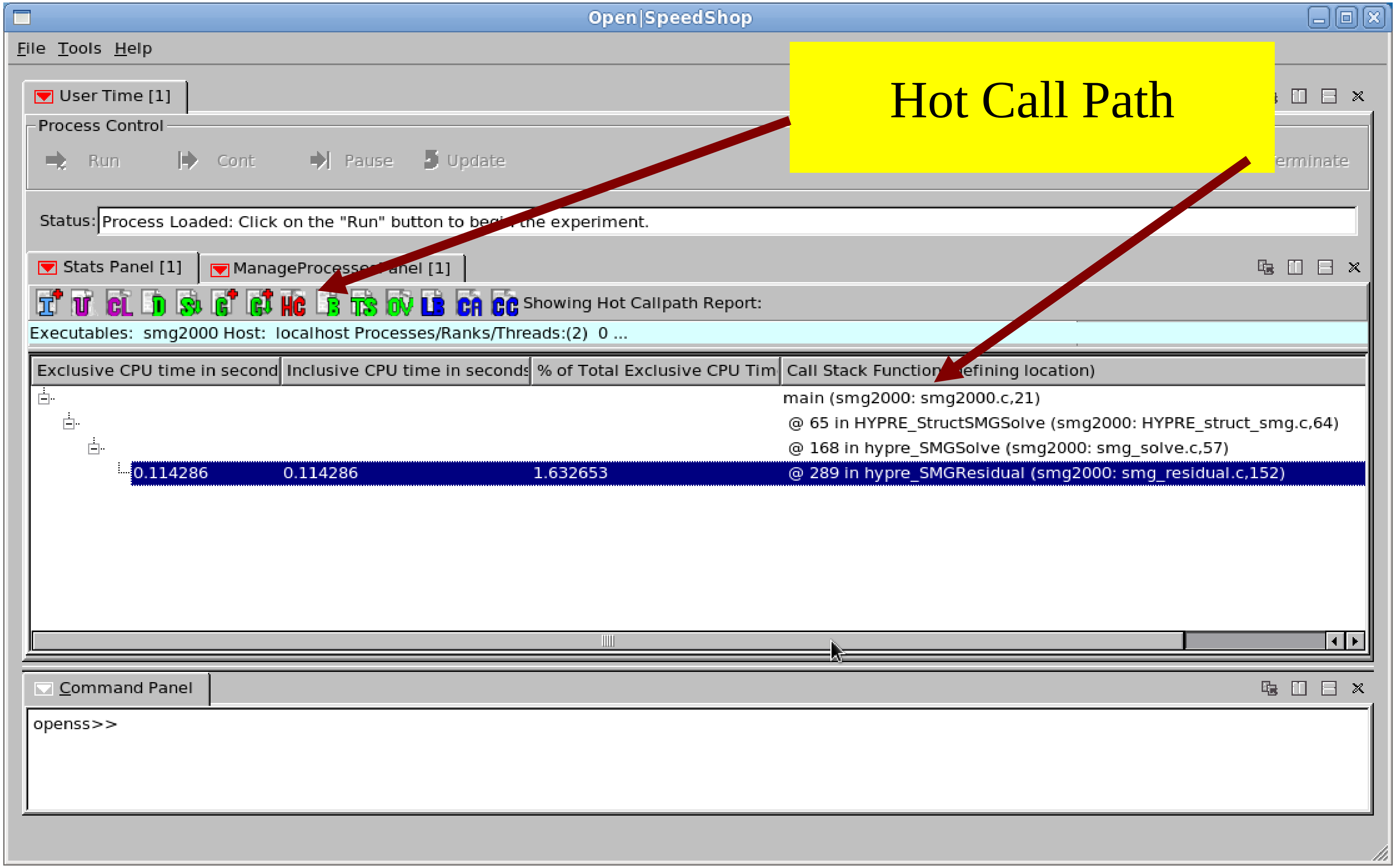
- **Exclusive raw data in left column**



Interpreting Context data

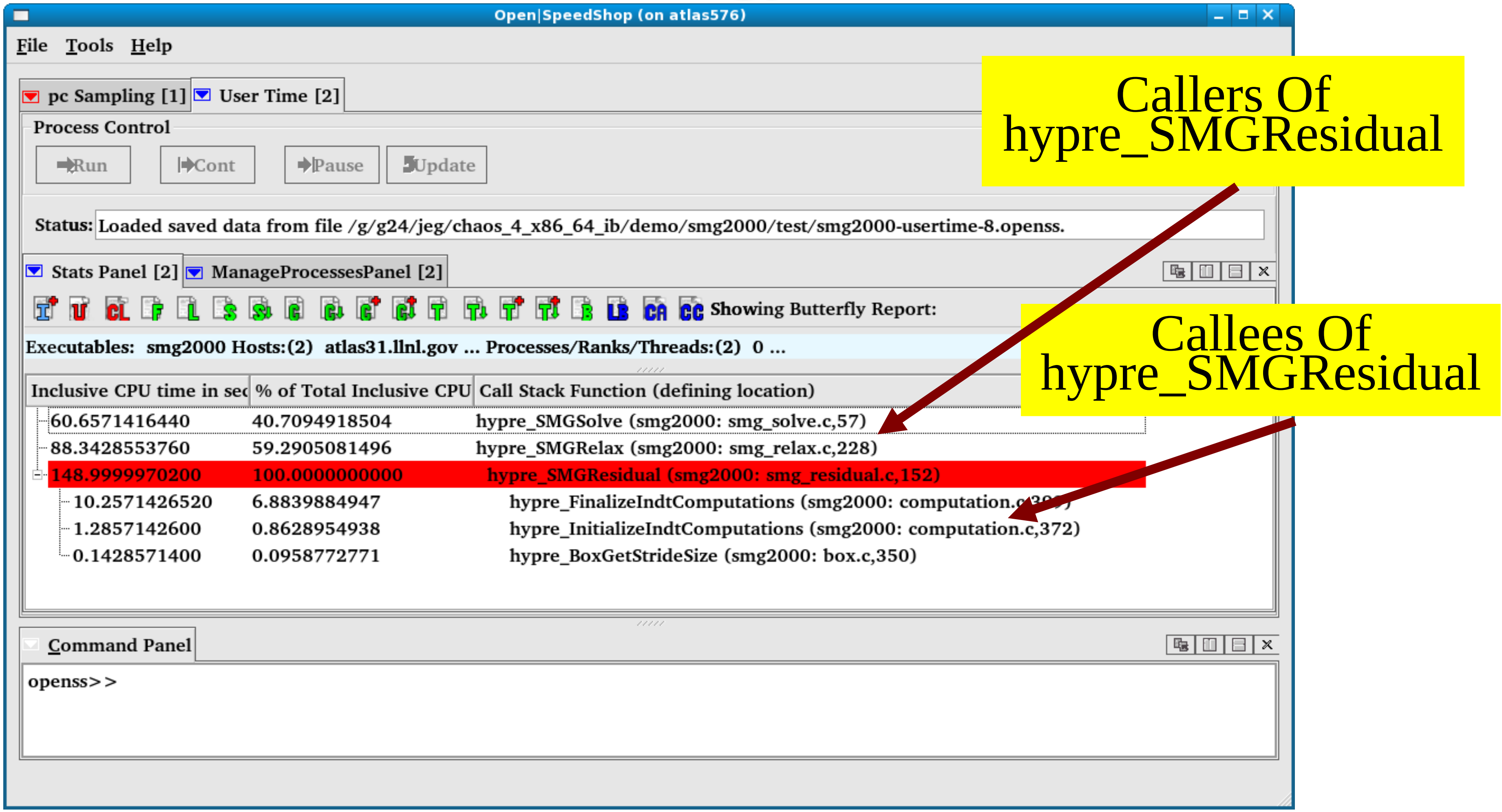
- **Inclusive vs. exclusive times**
  - If similar: child executions are insignificant
    - May not be useful to profile below this layer
  - If inclusive time >> exclusive time:
    - Execution mostly focus on children
- **Butterfly analysis**
  - Should be done on “suspicious” functions
  - Shows split of time in callees and callers

Call Stack / Stack Traces Views



Butterfly View

- **Similar to well known gprof tool**



Summary / Profiling

- **Profiling provides high-level information on an application's execution behavior**
  - Flat profiles show computational intensity
  - Varying granularity
  - Question: does this match intuition?
  - O|SS execution: osspcsamp "<app>"
- **Adding context**
  - Inclusive vs. exclusive timings
  - Caller/Callee data: butterfly views
  - Full context: stack traces or call stacks
  - OSS execution: ossusertime "<app>"

Open | SpeedShop™

Section 4
How to Relate Data to Architectural Properties?

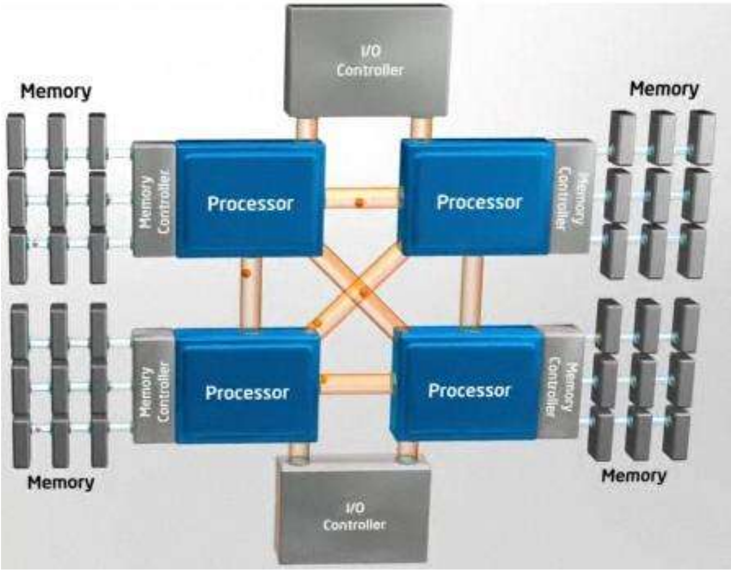
How to Analyze the Performance of Parallel Codes 101
A Case Study with Open|SpeedShop

What Timing Alone Doesn't Tell

- **Timing information shows you where you spend your time,**
  BUT: not why you spend time there
  - Are computational intensive parts efficient?
  - Which resources constrain execution?
- **Next: investigate App/HW interactions**
  - Efficient use of hardware resources
  - Architectural units that are stressed
- **Often platform dependent**
  - Cause of missing performance portability
  - Tuning to architectural parameters

The Memory System

- **Modern memory systems are complex**
  - Deep hierarchies
  - NUMA behavior
  - Streaming/prefetching
- **Key: locality**
- **Information to look for**
  - Read/Write intensity
  - Prefetch efficiency
  - Cache miss rates at all levels
  - TLB miss rates
  - NUMA overheads



Other Architectural Features

- **Computational intensity**
  - Cycles per instructions (CPI)
  - Number of floating point instructions
- **Branches**
  - Number of branches taken (pipeline flushes)
  - Miss-speculations
- **System-wide information**
  - I/O busses
  - Network counters
  - Power/Temperature sensors
  - BUT: not clear how to related to a process

Performance Counters

- **Most CPUs provide a set of counters**
  - Hardware to observe low level events
  - Architecture dependent
  - Semantic mapping hard
- **Newer components also include counters**
  - Network cards and switches
  - Environmental sensors
- **Recommended: access through PAPI**
  - Abstraction for system specific layers
  - API for tools & simple runtime tools
  - http://icl.cs.utk.edu/papi/

Directly using PAPI

- **“papiex” provides end-to-end data**
  - Use: papiex “<app>”
  - Overview printed after termination
  - More options: papiex –h
- **Useful to get a quick overview**
  - Enables basic classification of codes
  - Memory vs. computational intensive
  - Find relevant counters
- **Missing information**
  - Relation to source code
  - Execution context

The HWC Experiment

- **Provides access to hardware counters**
  - Detailed, low level information
  - Examples: cache & TLB misses, bus accesses
  - Time spent in a function and its children
- **Run until counter reaches threshold**
  - Record PC location and reset
  - User selects counter to track and sets threshold
  - Ideal threshold depends on application and how often it invokes the observed counter
- **Two versions of the experiment:**
  - HWC = flat hardware counter profile
  - HWCtime = profile with context / stacktraces

Selecting Counter/Threshold

- **Open|SpeedShop supports …**
  - Non-derived PAPI presets
    - All non derived events reported by “papi_avail –a”
    - Also reported by running osshwc[time]
  - All native events
    - Architecture specific
    - Names listed in PAPI documentation
- **Thresholds depend on application**
  - Overhead vs. accuracy
  - Rare events need small threshold
  - Frequent events need high threshold
  - Takes experience and/or trial & error

Running HWC / HWCtime

Offline hwc Experiment – smg2000

Option 1: Basic syntax

osshwc[time] “smg2000 –n 50 50 50” <counter> <threshold>

Option 2: Explicit Syntax

setenv OPENSS_HWC_EVENT <counter>

setenv OPENSS_HWC_THRESHOLD <threashold>

openss –offline –f “smg2000 –n 50 50 50” hwc[time]

- **Available counter**
  - Any counter reported by papi_avail or osshwc
  - Must be marked as “available” and “not derived”
  - Naïve counters listed in PAPI documentation

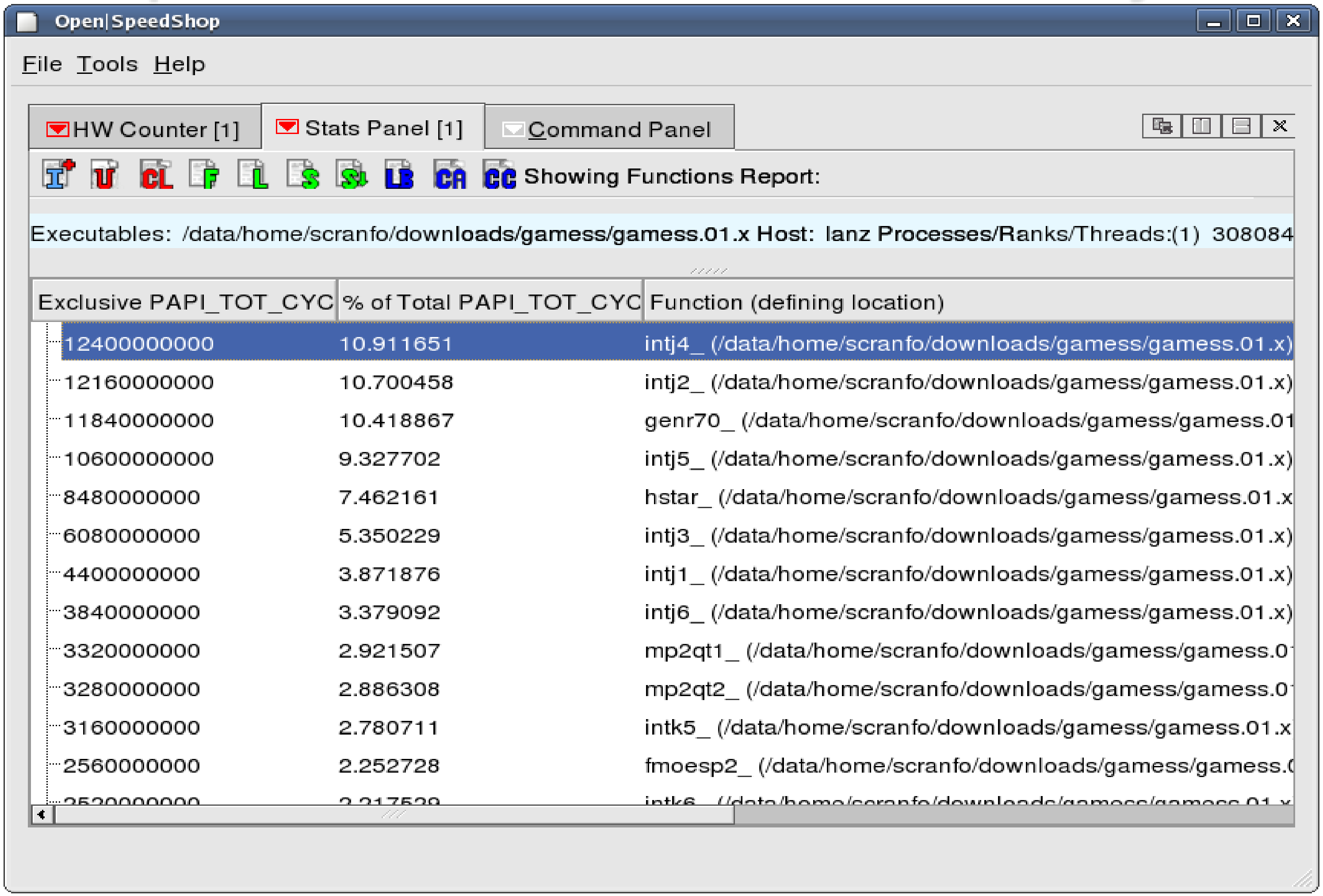
Examples of Counters

PAPI Name	Description	Threshold
PAPI_L1_DCM	L1 data cache misses	high
PAPI_L2_DCM	L2 data cache misses	high/medium
PAPI_L1_DCA	L1 data cache accesses	high
PAPI_FPU_IDL	Cycles in which FPUs are idle	high/medum
PAPI_STL_ICY	Cycles with no instruction issue	high/medium
PAPI_BR_MSP	Mispredicted branchnes	medium/low
PAPI_FP_INS	Number of floating point instructions	high
PAPI_LD_INS	Number of load instructions	high
PAPI_VEC_INS	Number of vector/SIMD instructions	high/medium
PAPI_HW_INT	Number of hardware interrupts	low
PAPI_TLB_TL	Number of TLB misses	low

Note: Threshold indications are just rough guidance and depend on the application
Not all counters exist on all applications (run papi_avail to find out)

Viewing HWC Data

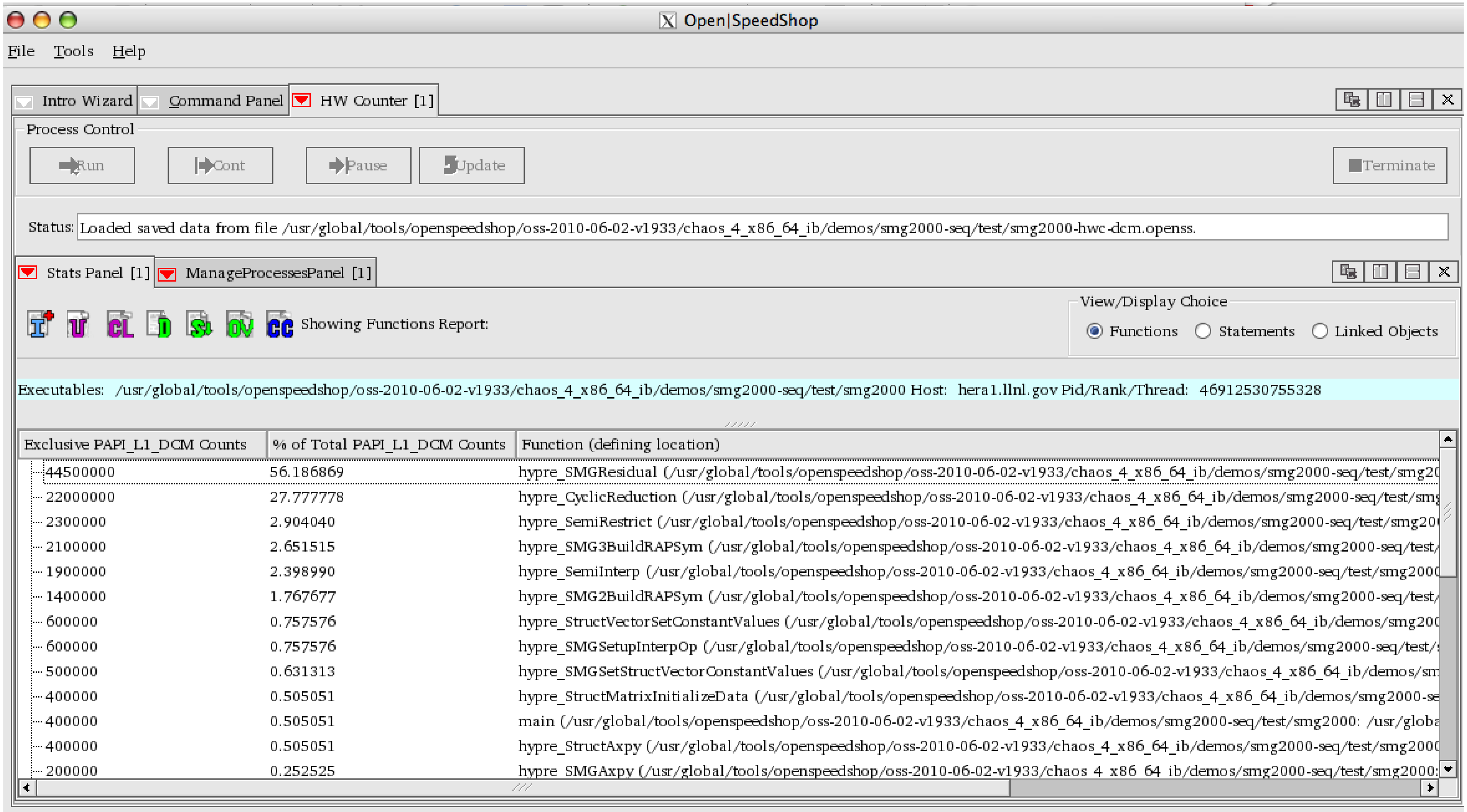
- **hwc Experiment – Default View – Counter=cycles**



Interpreting Memory Data

- **Typical question:**
  - How well is my code exploiting locality?
  - What is my cache behavior?
- **Step 1: Look for cache misses**
  - Counter in PAPI presets
  - Event: PAPI_L1_DCM
  - Threshold 100,000
- **Run experiment**
  - osshwc smg2000 PAPI_L1_DCM 100000
  - Creates new database with miss data

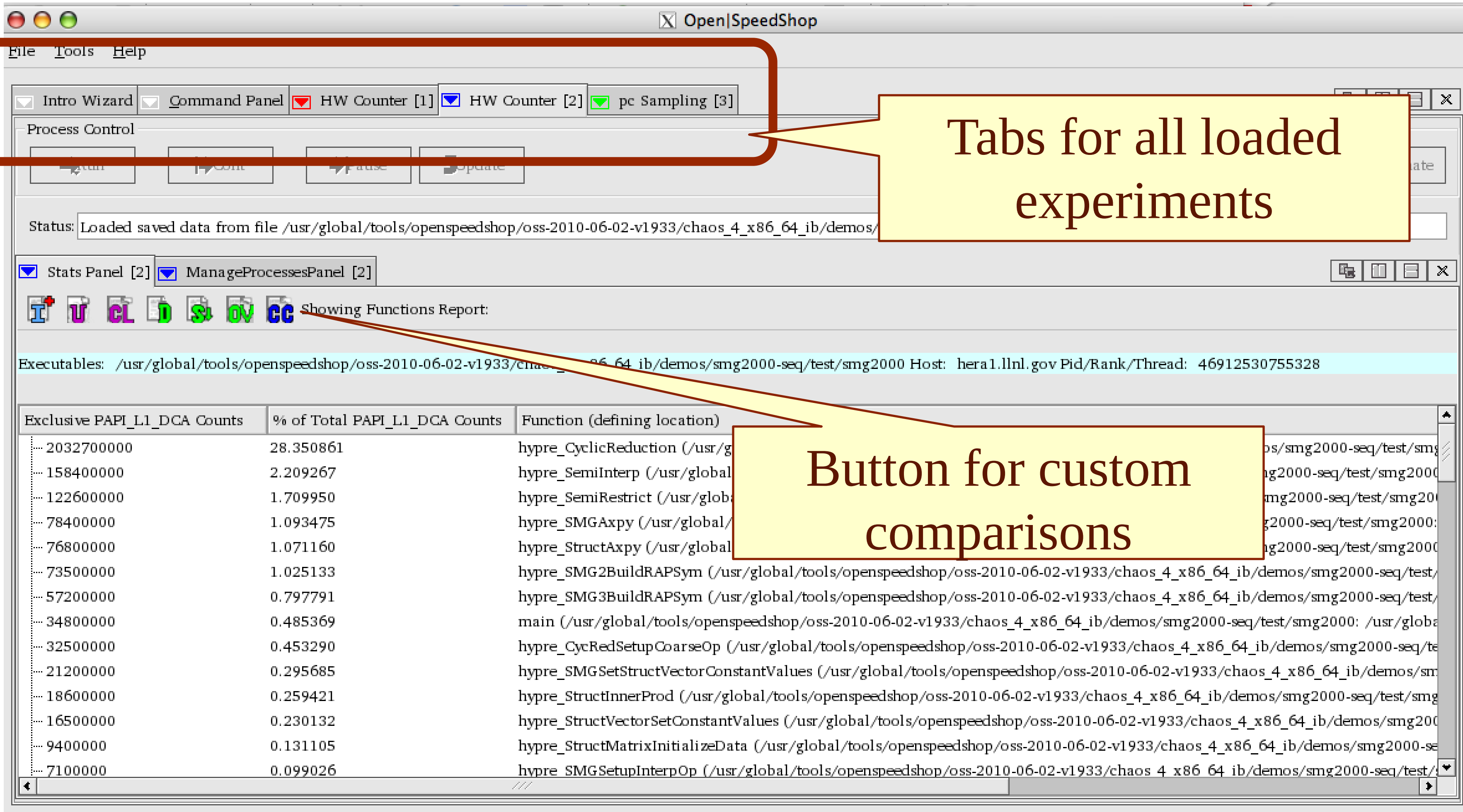
Viewing L1 Miss Data



Interpreting L1 miss data

- **Miss numbers don’t tell much**
  - Is the number of misses good or bad?
  - Is it in a relevant piece of code?
- **Need additional information**
  - Combine with flat profiles
  - Get number of cache accesses
    - Miss rates
- **Need to run additional experiments**
  - osspcsamp smg2000
  - osshwc smg2000 PAPI_L1_DCA 100000

Viewing Multiple Experiments



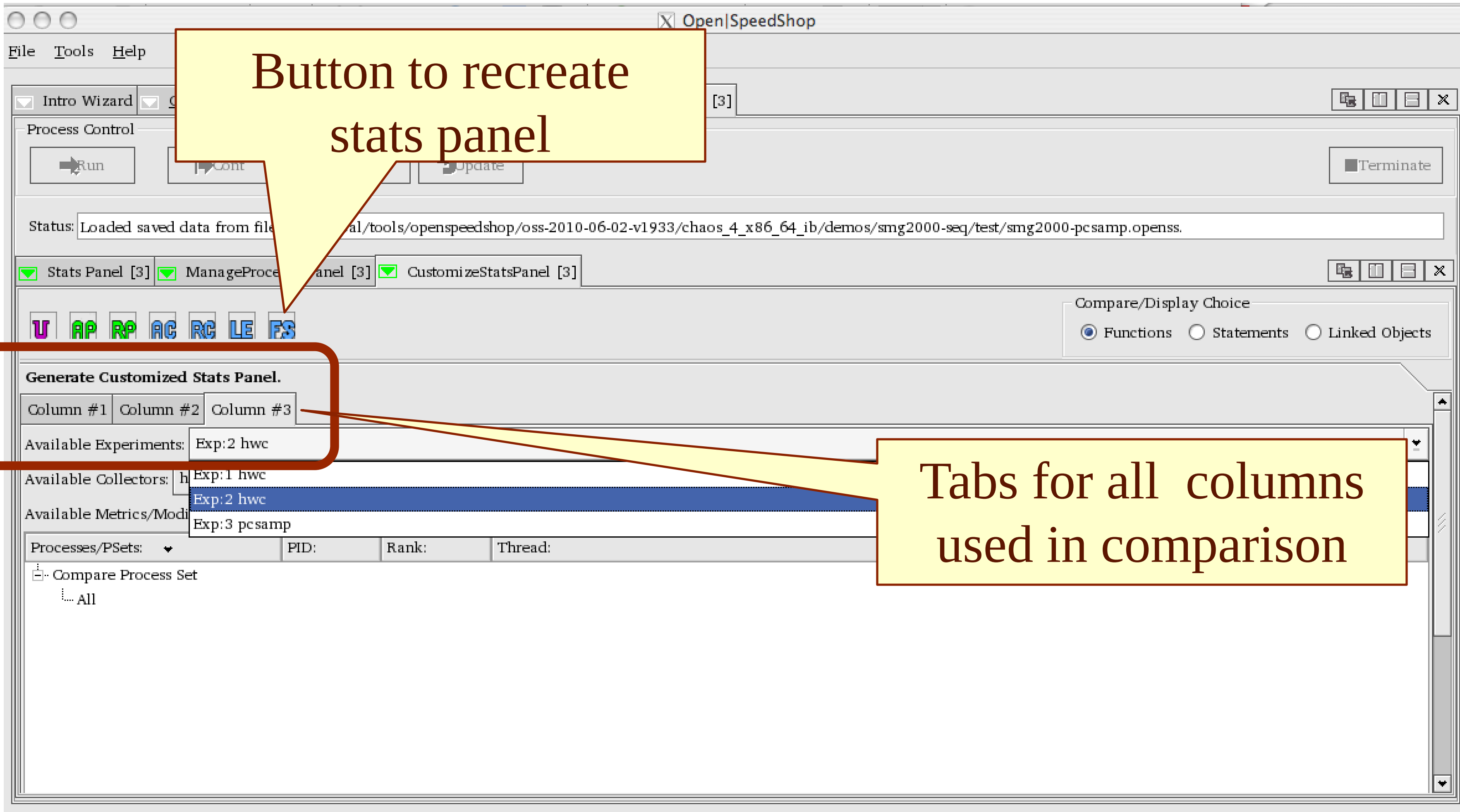
Tabs for all loaded experiments

Button for custom comparisons

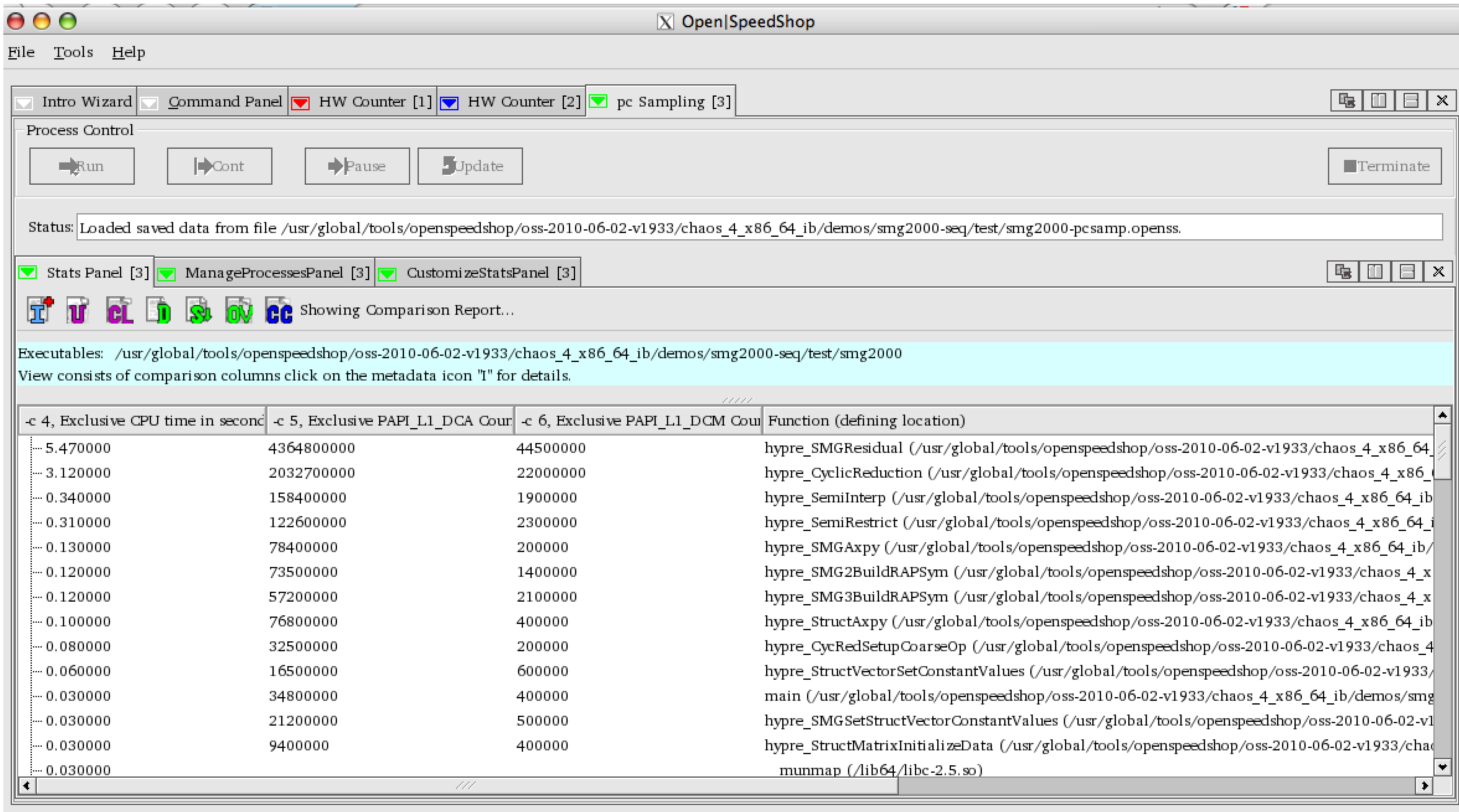
Custom Comparisons

- **Goal: direct comparison of measurements**
  - Flat profile / time spent
  - Number of cache accesses
  - Number of cacha misses
- **O|SS concept: custom comparison**
  - Arbitrary number of metrics
  - Placed in separate "columns"
  - Each row shows multiple metrics for one location
- **Creating custom comparisons**
  1. Load all three experiments
  2. Chose "CC" in one of the views
  3. Add columns and select different experiments in each
  4. Click on "Focus Stats Panel" ("FS")

Custom Comparisons in O|SS



Comparison Results



Summary / Hardware Counters

- **Hardware counter provide low level data**
  - Architecture-level information
  - Interpretation often machine specific
- **Selection requires some architecture knowledge**
  - Meaning of counters
  - Correct threshold
- **O|SS's HWC experiment**
  - Select any PAPI event and threshold
  - Display similar to profiles
- **Interpreting HWC experiments**
  - Use flat profiles as baslines
  - Raw measurements often not useful
  - Compare measurements with other counters

Open | SpeedShop™

Section 5
How to Find I/O Bottlenecks?

How to Analyze the Performance of Parallel Codes 101
A Case Study with Open|SpeedShop

Need for Understanding I/O

- **Could be significant percentage of execution:**
  - Checkpoint and viz I/O frequency
  - I/O pattern in application: N-to-1 / N-to-N / …
  - Kind of application: data intensive vs. traditional HPC
  - File system (Lustre, Panfs, NFS, GPFS)
  - I/O libraries – MPI-IO, hdf5, PLFS, etc.
  - Other jobs stressing the I/O nodes on a system
- **Obvious candidates to explore first**
  - Use parallel file system
  - Optimize for I/O pattern
  - Match checkpoint I/O frequency to MTBI of system
  - Use appropriate libraries (such as iobuf on Cray XTs)

I/O Performance Example

- **Application:**
  OOCORE benchmark from DOD HPCMO
  - Out-of-core SCALPACK benchmark from UTK
  - Can be configured to be disk I/O intensive
  - Characterizes a very important class of HPC application involving the use of Method of Moments (MOM) formulation for investigating Electromagnetics (e.g. Radar Cross Section, Antenna design)
  - Solves dense matrix equations by LU, QR or Cholesky
  - Reference: *Benchmarking OOCORE*, and out-of-core Matrix Solver, By Drs. Samuel B. Cable and Eduardo D'Azevedo

Why use this example

- **Used by HPCMO to evaluate I/O system scalability**
- **For our needs this application or similar out-of-core dense solver benchmarks help to point out importance of the following in performance analysis**
  - I/O overhead minimization
  - Matrix Multiply kernel – possible to achieve close to peak performance of the machine if tuned well
  - ‘blocking’ very important to understand for modern processors (and HPC systems) with deep memory hierarchies

Execution on Multi-Core Cluster

INPUT: testdriver.in

ScaLAPACK out-of-core LU,QR,LL factorization input file

testdriver.out

6	device out
1	number of factorizations
LU	factorization methods -- QR, LU, or LT
1	number of problem sizes
31000	values of M
31000	values of N
1	values of nrhs
9200000	values of Asize
1	number of MB's and NB's
16	values of MB
16	values of NB
1	number of process grids
4	values of P
4	values of Q

Nodes: Quad-Core/Quad-Sockets

Output from run on 16 cores

TIME	M	N	MB	NB	NRHS	P	Q	Fact/Solve Time		Error	Residual
WALL	31000	31000	16	16	1	4	4	1842.20	1611.59	4.51E+15	1.45E+11

DEPS = 1.110223024625157E-016

sum(xsol_i) = (30999.9999999873,0.000000000000000E+000)

sum |xsol_i - x_i| = (3.332285336962339E-006,0.000000000000000E+000)

sum |xsol_i - x_i|/M = (1.074930753858819E-010,0.000000000000000E+000)

sum |xsol_i - x_i|/(M*eps) = (968211.548505533,0.000000000000000E+000)

Observe:

1) LU Fact time (Lustre)= 1842 secs; LU Fact time (NFS) = 2655 secs (delta t= 813 secs)

2) Application built with Intel 11.1, MVAPICH 1.1, mkl 11.1, BLACS 1.1

Investigate File system Impact with OSS:

ossio "/srun -N 1-n 16 ./testzdriver-std"

Filesystem: NFS vs. Lustre

- **Step 1: Understand difference between file systems**
  - Execute code on NFS & Lustre
  - Same platform / different target directory
- **Two Open|SpeedShop experiments**
  - Running: ossio “oocore”
  - Rename database between runs
- **Analysis**
  - Look at load balance information
  - Compare the maximal runtime per rank

Results

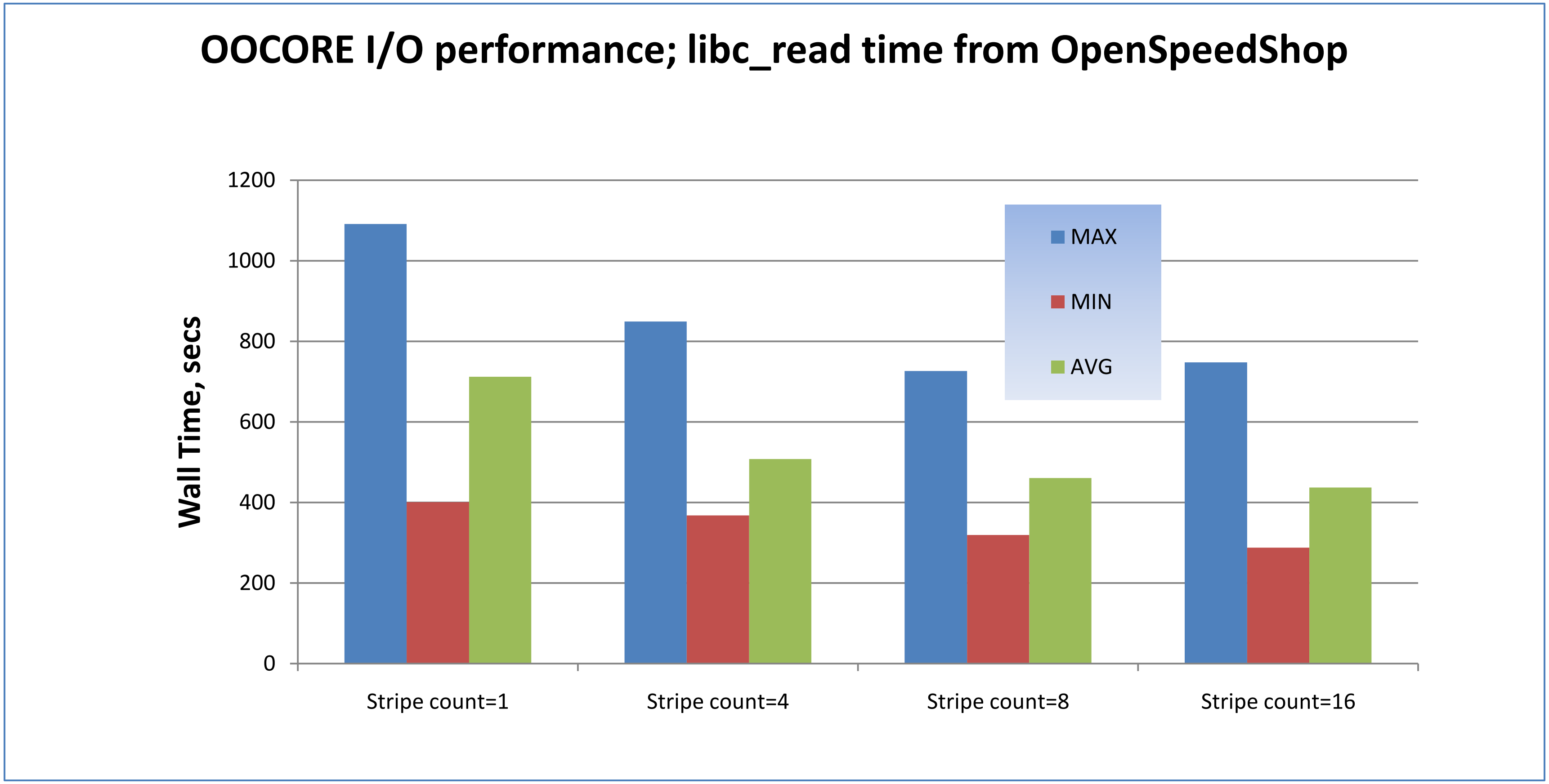
NFS RUN			
Min t (secs)	Max t (secs)	Avg t (secs)	call Function
1102.380076	1360.727283	1261.310157	__libc_read(/lib64/libpthread-2.5.so)
31.19218	99.[illegible]44468	49.01867	__libc_write(/lib64/libpthread-2.5.so)
0.162285	0.30[illegible]41	0.241362	llseek(/lib64/libpthread-2.5.so)
0.001505	0.0298[illegible]5	0.020403	__libc_open(/lib64/libpthread-2.5.so)
0.0005	0.0005	0.0005	__libc_open64(/lib64/libpthread-2.5.so)
0.000393	0.002367	0.001374	__libc_close(/lib64/libpthread-2.5.so)

LUSTRE RUN			
Min t (secs)	Max t (secs)	Avg t (secs)	call Function
368.898283	847.919127	508.658604	__libc_read(/lib64/libpthread-2.5.so)
6.27036	7.896153	6.850897	__libc_write(/lib64/libpthread-2.5.so)
0.646541	0.646541	0.646541	llseek(/lib64/libpthread-2.5.so)
0.06473	0.079408	0.072694	__libc_open(/lib64/libpthread-2.5.so)
0.00194	0.012322	0.007334	__libc_open64(/lib64/libpthread-2.5.so)
0.000548	0.111174	0.016195	__libc_close(/lib64/libpthread-2.5.so)

The run time difference 75% of 854 secs is mostly I/O:
(1360+99) – (847 +7) = 605 secs

OpenSpeedShop IO-experiment used to identify optimal *lfs* striping

(from load balance view (max, min & avg) for 16 way parallel run)



Additional I/O Capabilities

- **Extended I/O Tracing (iot experiment)**
  - Records each event in chronological order
  - Collects Additional Information
    - Function Parameters
    - Function Return Value
- **When to use extended I/O tracing?**
  - When you want to trace the exact order of events
  - When you want to see the return values or bytes read or written.

Summary

- **I/O Collectors**
  - Intercept All Calls to I/O Functions
  - Record Current Stack Trace & Start/End Time
  - Can Collect Detailed Ancillary Data (IOT)
- **Trace experiments**
- **Collect large amounts of data**
- **Allows for fine-grained analysis**

Open | SpeedShop™

Section 6
How to Find Bottlenecks in Parallel Codes?

How to Analyze the Performance of Parallel Codes 101
A Case Study with Open|SpeedShop

Experiment Types

- **O|SS is designed to work on parallel jobs**
  - Support for threading and message passing
  - Focus here: parallelism using MPI
- **Sequential experiments**
  - Apply experiment/collectors to all nodes
  - By default display aggregate results
  - Optional select individual groups of processes
- **MPI tracing experiments**
  - Tracing of MPI calls
  - Similar to I/O tracing
  - Also available with and without parameters
    - mpi vs. mpit
  - OTF version

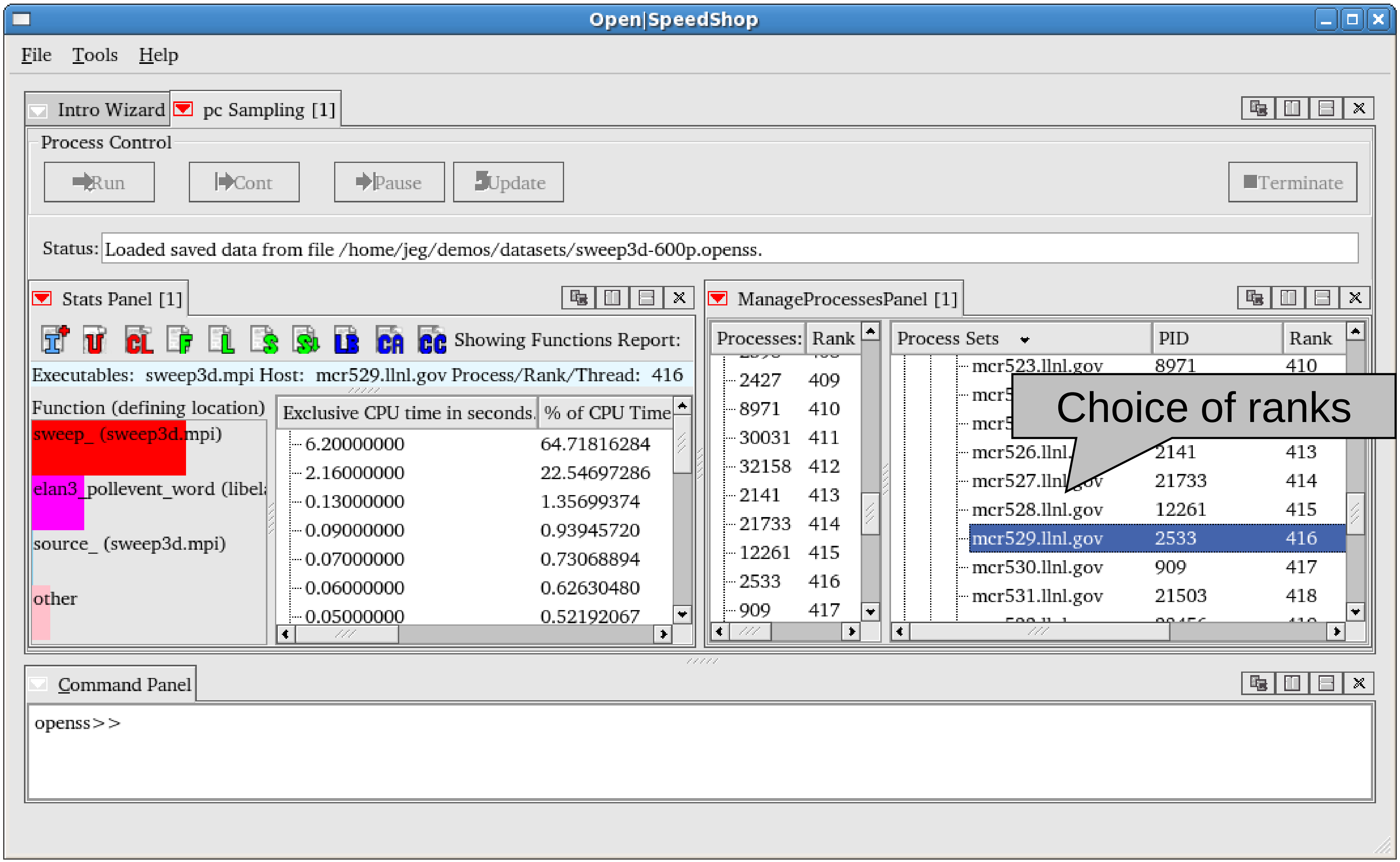
Integration with MPI

- **O|SS has been tested with a variety of MPIs**
  - Includes Open MPI, MPVAPICH, and MPICH-2
- **Identifying MPI tasks**
  - Online: through MPIR interface
  - Offline: through PMPI preload
- **Running with MPI codes**
  - Add MPI starter as part of the executable name
  - ossmpi “orterun –np 16 sweep3d.mpi”
  - osspcsamp “mpirun -np 4 sweep3d.mpi”
  - openss –offline –f “srun –N 4 –n 16 sweep3d.mpi” pcsamp
  - openss –online –f “orterun –np 16 sweep3d.mpi” usertime

Parallel Result Analysis

- **Default views**
  - Values aggregated across all ranks
  - Manually include/exclude individual processes
- **Rank comparisons**
  - Use Customize Stats Panel View
  - Create columns for process groups
- **Cluster Analysis**
  - Automatically create process groups of similar processes
  - Available from Stats Panel context menu

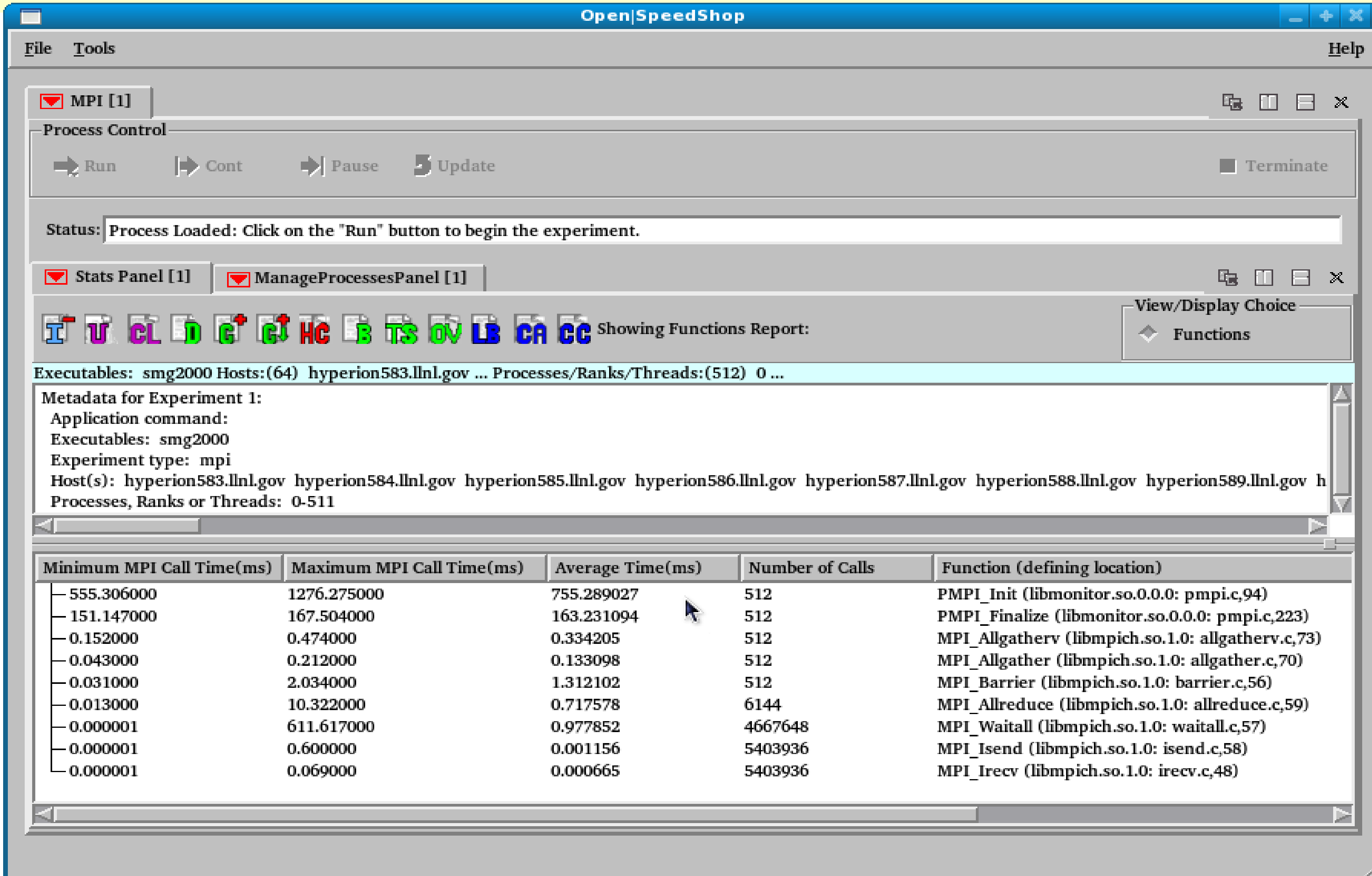
Viewing Results by Process



MPI Tracing

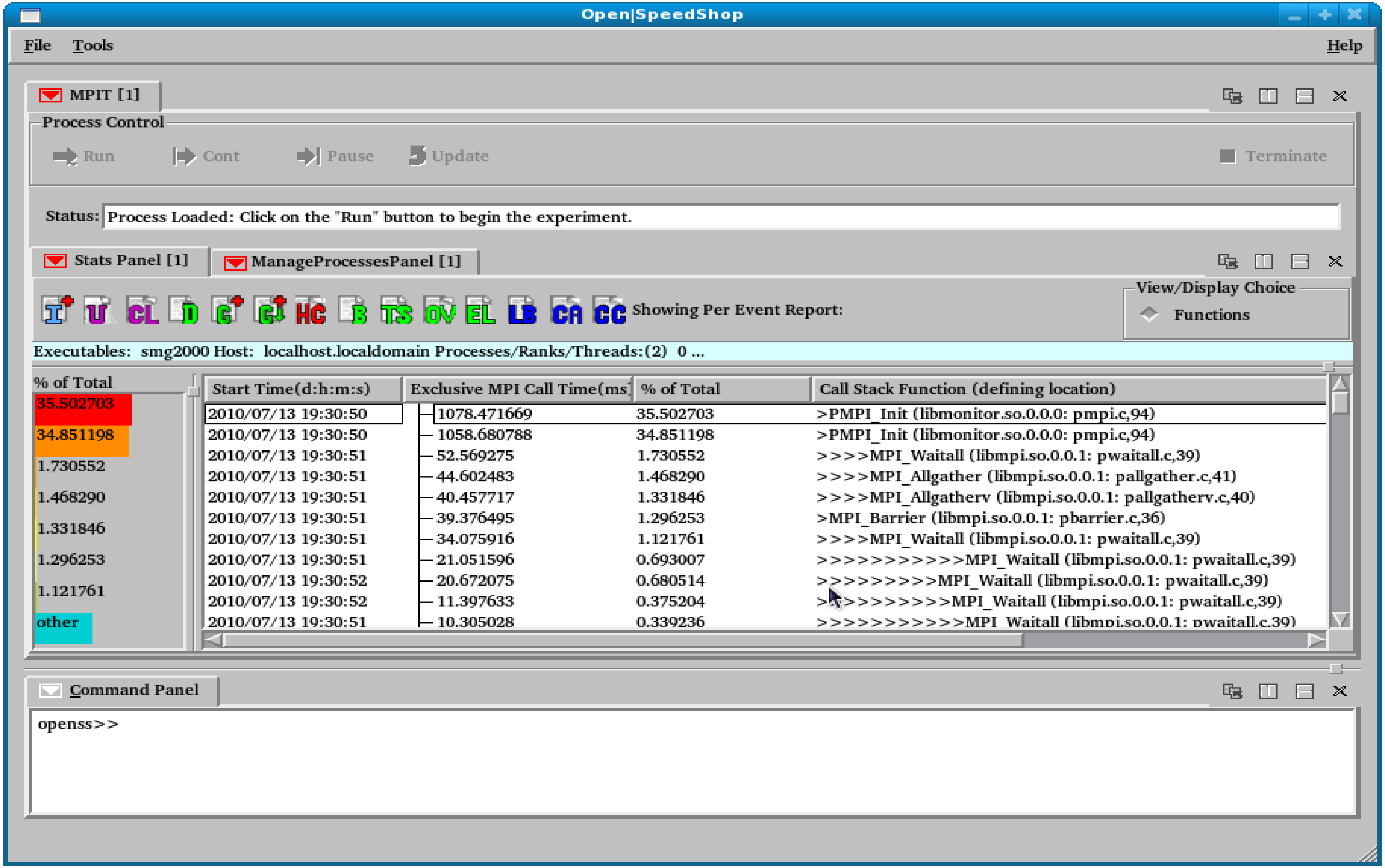
- **Similar to I/O tracing**
  - Record all MPI call invocations
  - By default: record call times (mpi)
  - Optional: record all arguments (mpit)
- **Equal events will be aggregated**
  - Save space in database
  - Reduce overhead
- **Public format:**
  - Full MPI traces in Open Trace Format (OTF)

Tracing Results: Default View

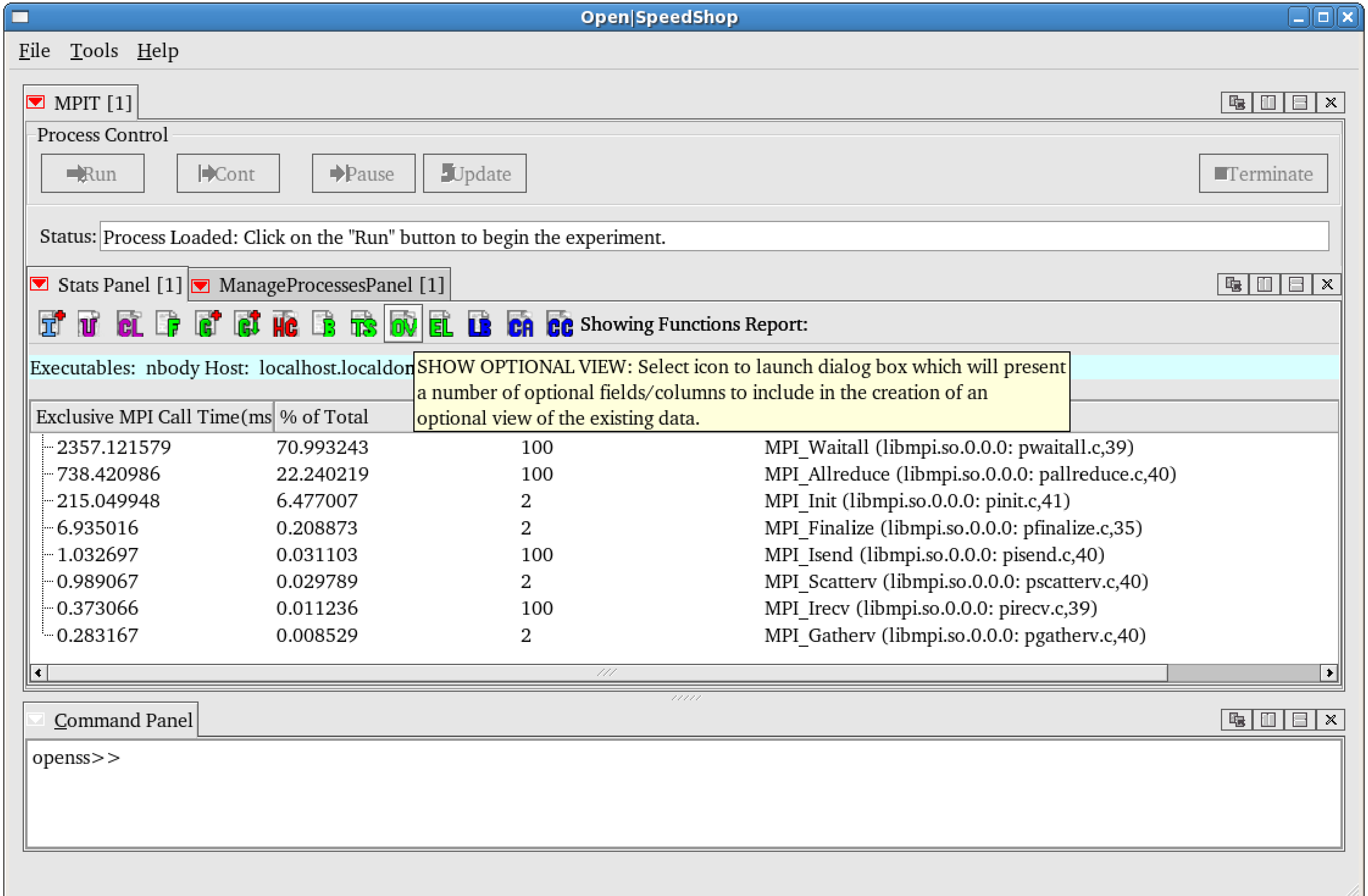


Minimum MPI Call Time(ms)	Maximum MPI Call Time(ms)	Average Time(ms)	Number of Calls	Function (defining location)
555.306000	1276.275000	755.289027	512	PMPI_Init (libmonitor.so.0.0.0: pmpi.c,94)
151.147000	167.504000	163.231094	512	PMPI_Finalize (libmonitor.so.0.0.0: pmpi.c,223)
0.152000	0.474000	0.334205	512	MPI_Allgatherv (libmpich.so.1.0: allgatherv.c,73)
0.043000	0.212000	0.133098	512	MPI_Allgather (libmpich.so.1.0: allgather.c,70)
0.031000	2.034000	1.312102	512	MPI_Barrier (libmpich.so.1.0: barrier.c,56)
0.013000	10.322000	0.717578	6144	MPI_Allreduce (libmpich.so.1.0: allreduce.c,59)
0.000001	611.617000	0.977852	4667648	MPI_Waitall (libmpich.so.1.0: waitall.c,57)
0.000001	0.600000	0.001156	5403936	MPI_Isend (libmpich.so.1.0: isend.c,58)
0.000001	0.069000	0.000665	5403936	MPI_Irecv (libmpich.so.1.0: irecv.c,48)

Tracing Results: Event View (default)



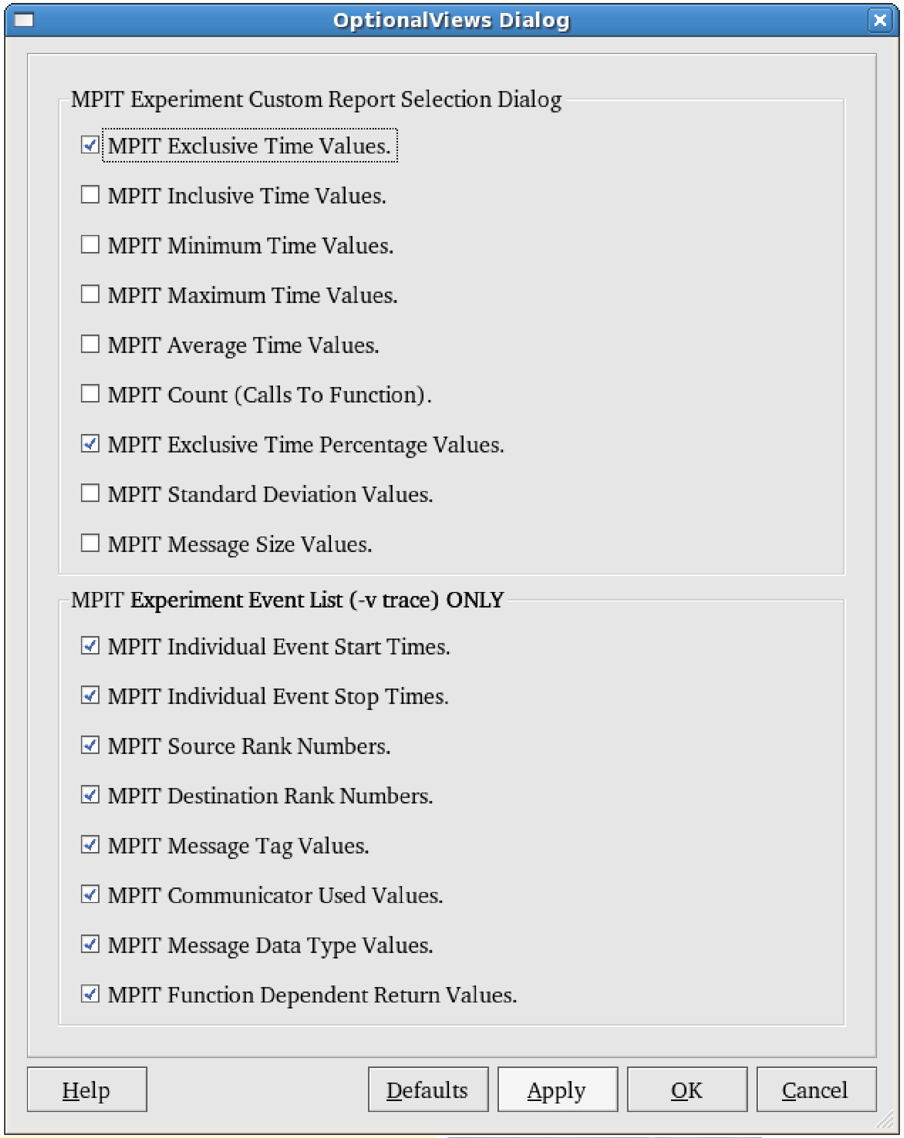
Tracing Results: Creating Event View



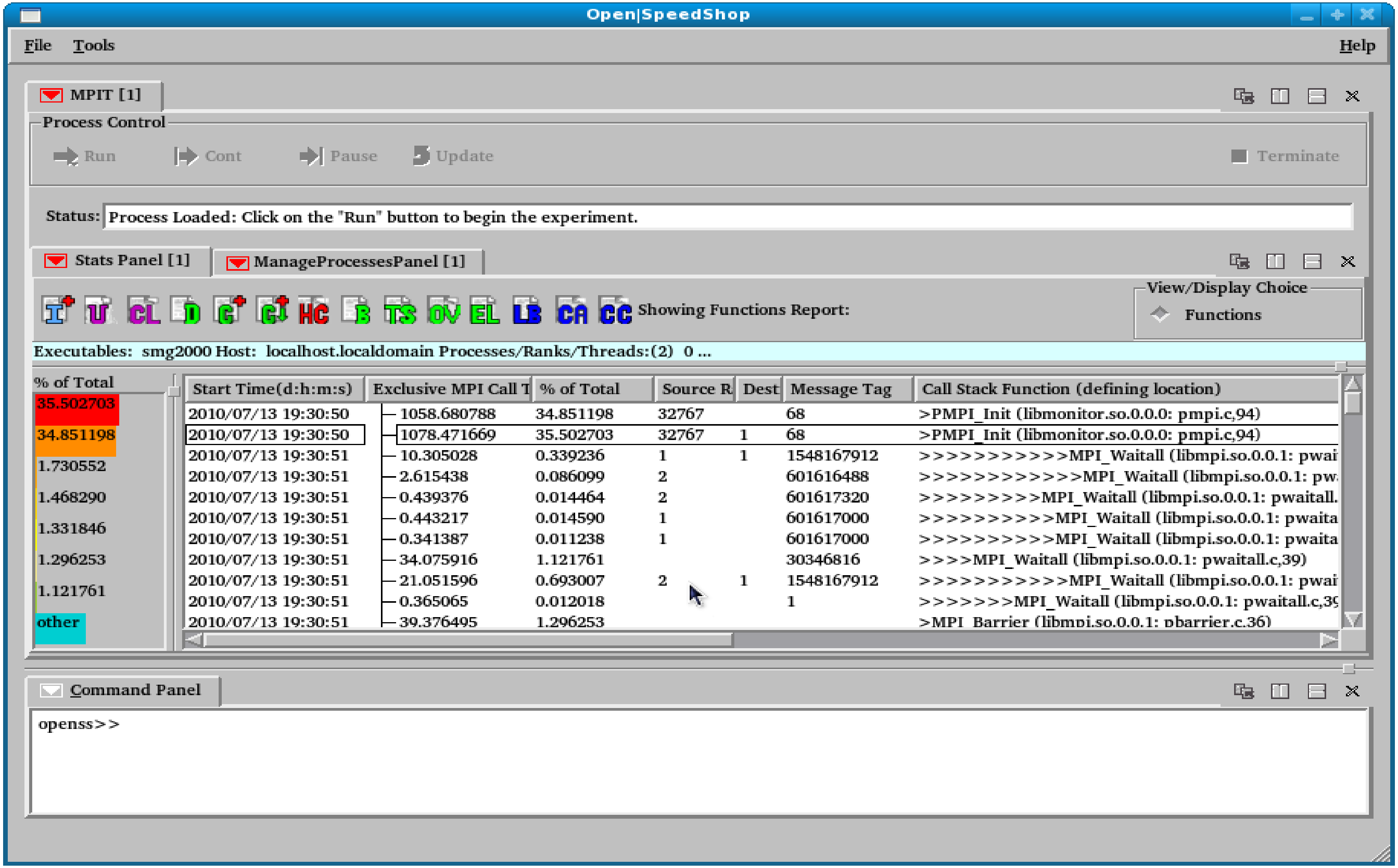
Tracing Results: Creating Specialized Event View

Use the Optional Views Dialog box to choose the performance metrics to be displayed in the StatsPanel and click OK

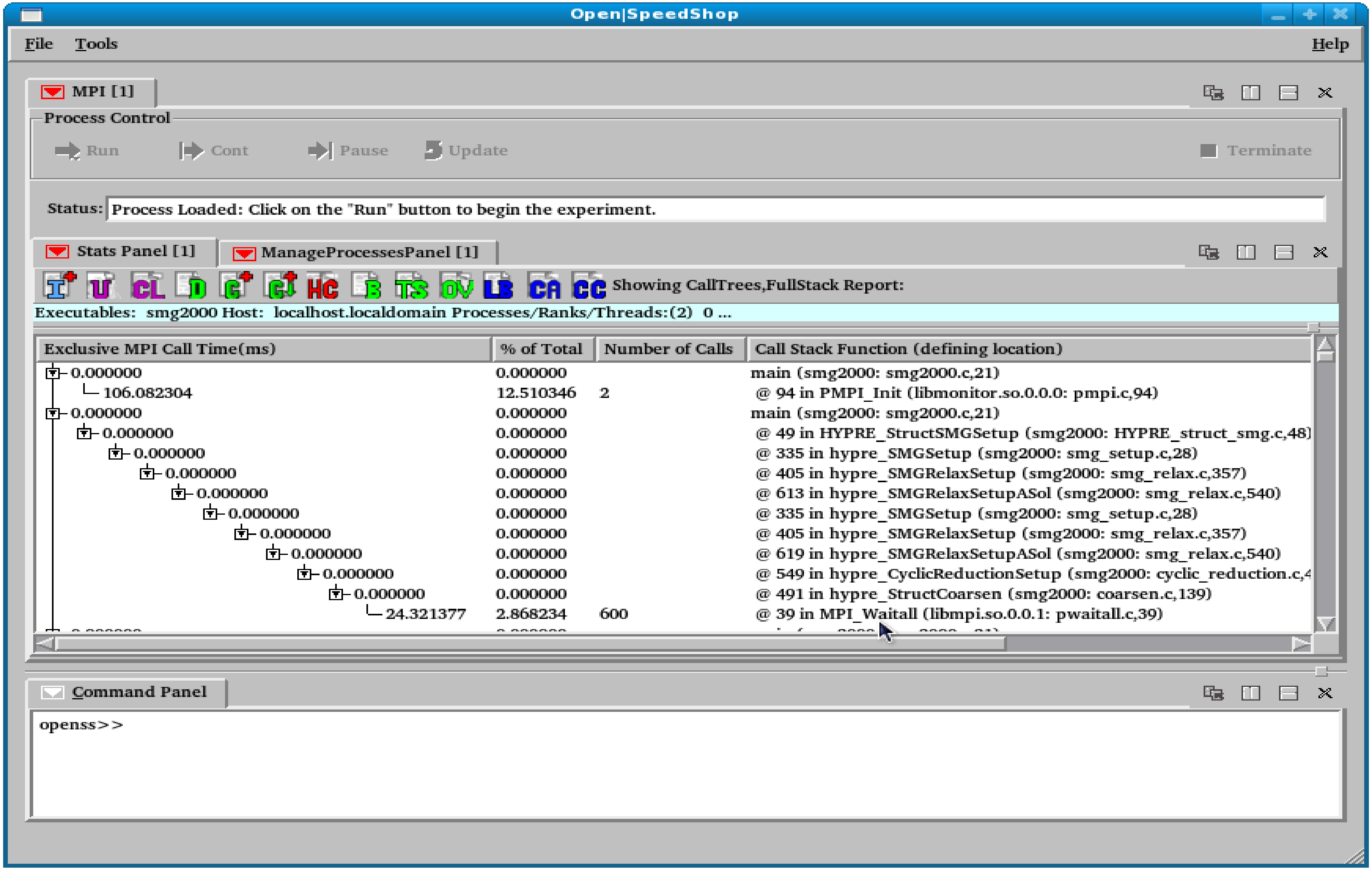
Clicking OK will regenerate the StatsPanel with the new metrics displayed



Tracing Results: Specialized Event View



Results / Show: Callstacks

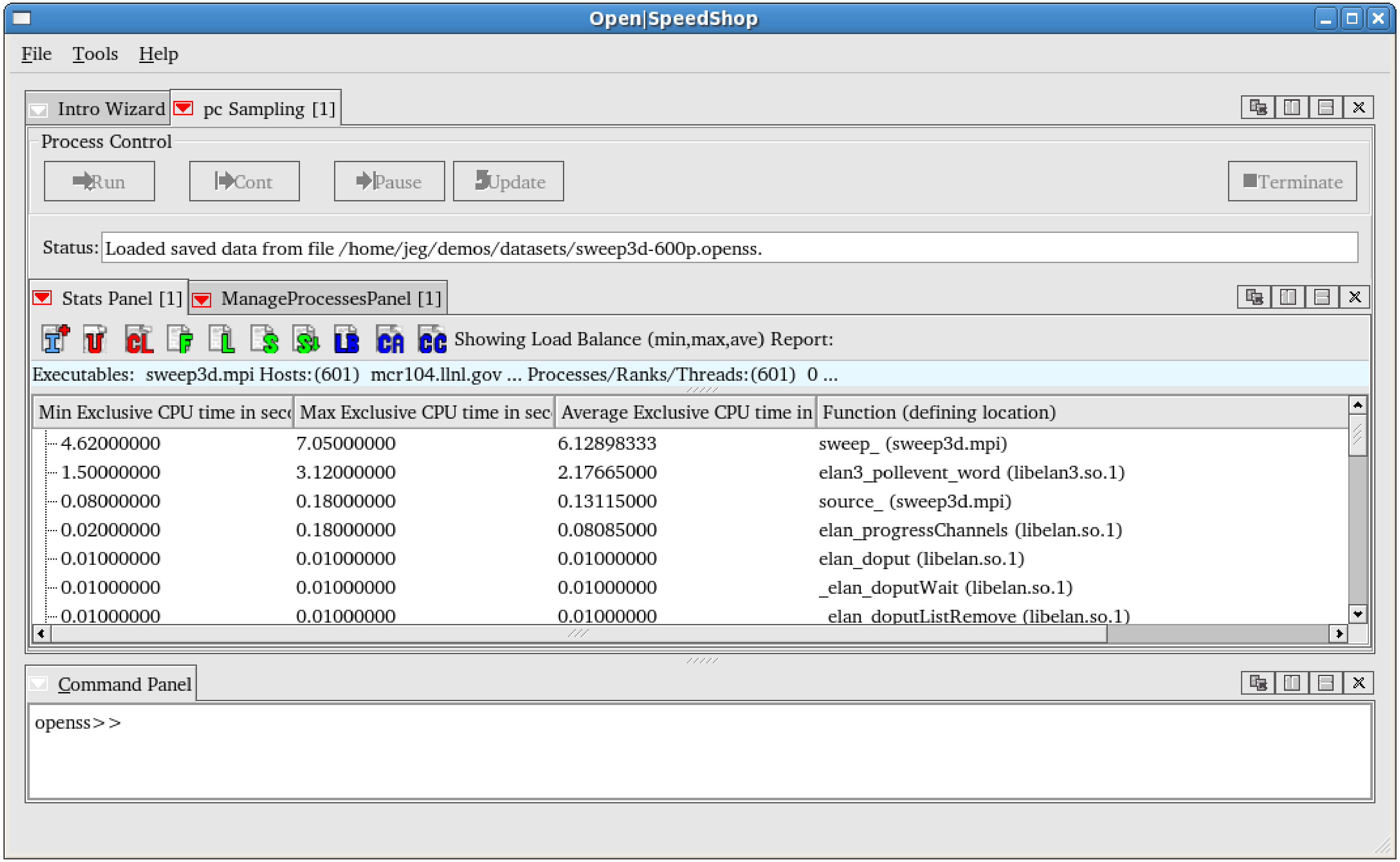


Predefined Analysis Views

- **O|SS provides common analysis functions**
  - Designed for quick analysis of MPI applications
  - Create new views in the StatsPanel
  - Accessible through context menu or toolbar
- **Load Balance View**
  - Calculate min, max, average across ranks, processes or threads
- **Comparative Analysis View**
  - Use “cluster analysis” algorithm to group like performing ranks, processes, or threads.

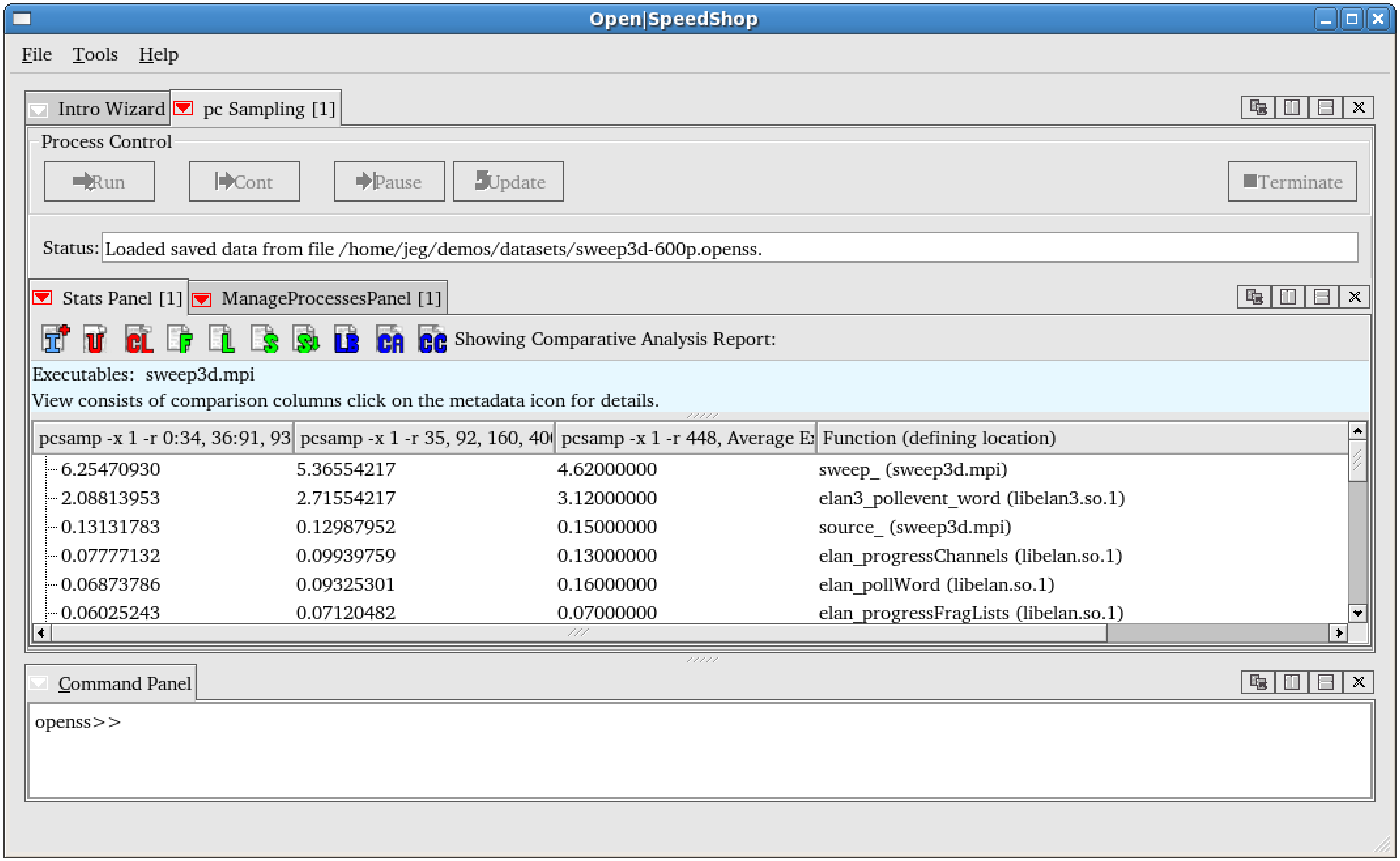
Quick Min, Max, Average View

- **Load Balance View: “LB” in Toolbar**

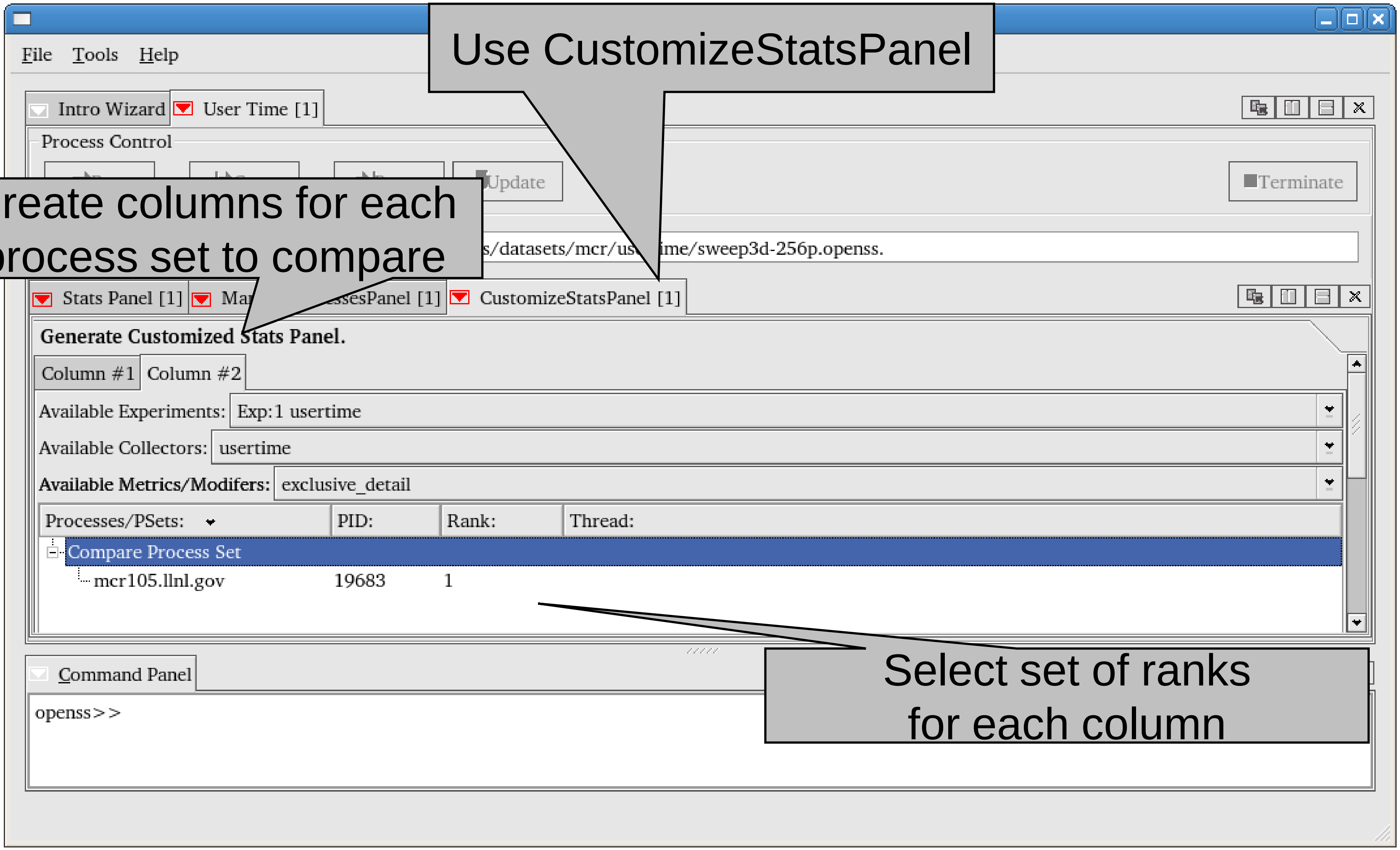


Comparative Analysis: Clustering Ranks

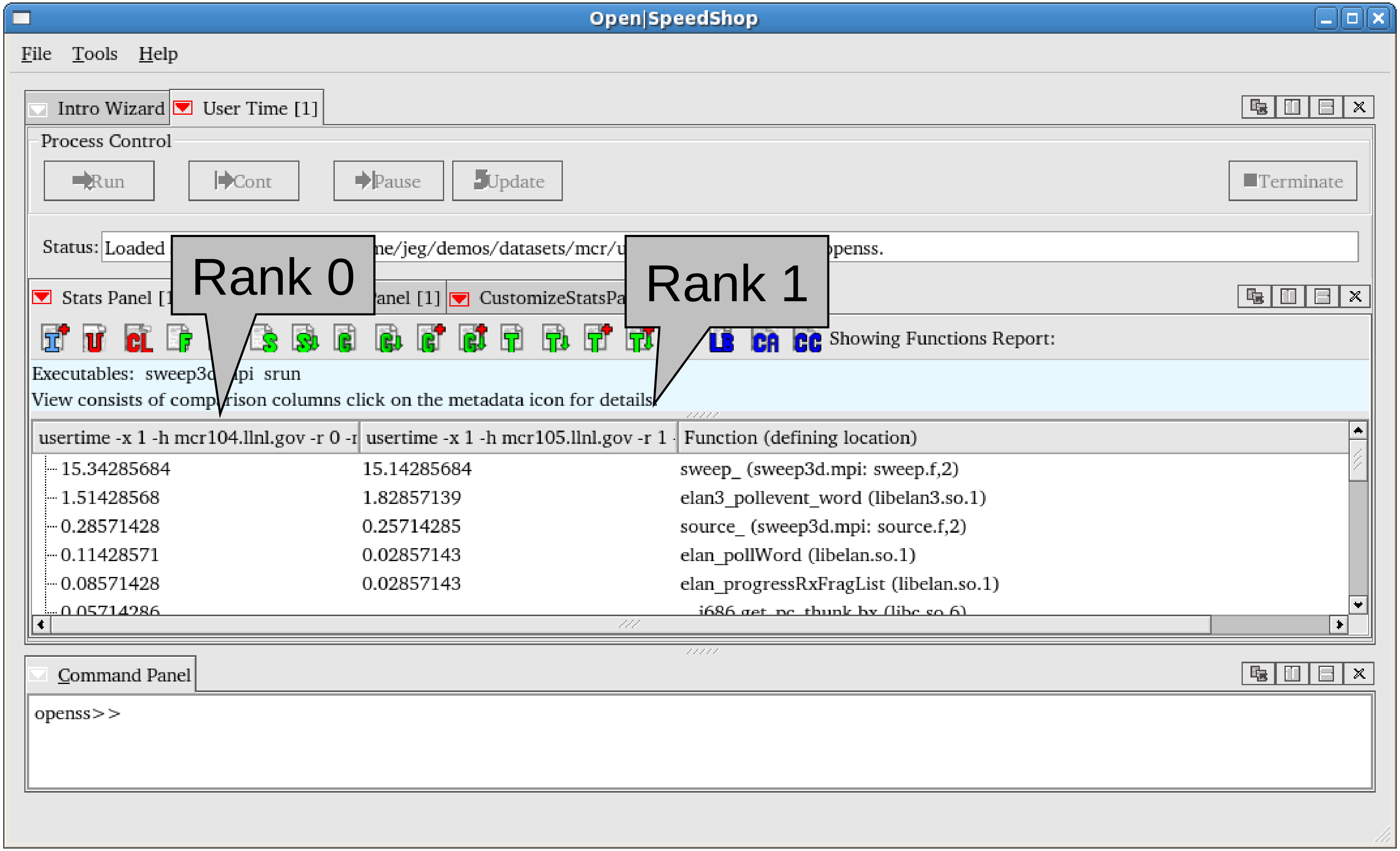
- **Comparative Analysis: “CA” in Toolbar**



Comparing Ranks (1)



Comparing Ranks (2)



Summary

- **Open|SpeedShop manages MPI jobs**
  - Works with multiple MPI implementations
  - Process control using MPIR interface (dynamic)
- **Parallel experiments**
  - Apply sequential collectors to all nodes
  - Specialized MPI tracing experiments
- **Results**
  - By default aggregated across results
  - Optional: select individual processes
  - Compare or group ranks & specialized views

Open | SpeedShop™

Section 7
How can I repeat this at Home?
What else can I do with O|SS?

How to Analyze the Performance of Parallel Codes 101
A Case Study with Open|SpeedShop

System Requirements

- **System architecture**
  - AMD Opteron/Athlon
  - Intel x86, x86-64, and Itanium-2
- **Operating system**
  - Tested on Many Popular Linux Distributions
    - SLES, SUSE
    - RHEL
    - Fedora Core, CentOS
    - Debian, Ubuntu
    - Varieties of the above

Getting the Source

- **Sourceforge Project Home**
  - http://sourceforge.net/projects/openss
- **CVS Access**
  - http://sourceforge.net/cvs/?group_id=176777
- **Packages**
  - Accessible From Project Home Download Tab
- **Additional Information**
  - http://www.openspeedshop.org/

Build tool overview

- install.sh
  - Bash script used to build the components that Open|SpeedShop uses and Open|SpeedShop
  - First will check to see if you have the correct supporting software installed on your system
    - If not, will stop and ask you if you want to continue
  - Will, optionally, then build and install all the prerequisite packages and also Open|SpeedShop itself
  - Can build and install one component at a time.
  - Builds and installs single or groups of components so that the next components use the previous components.

Post-Installation Setup

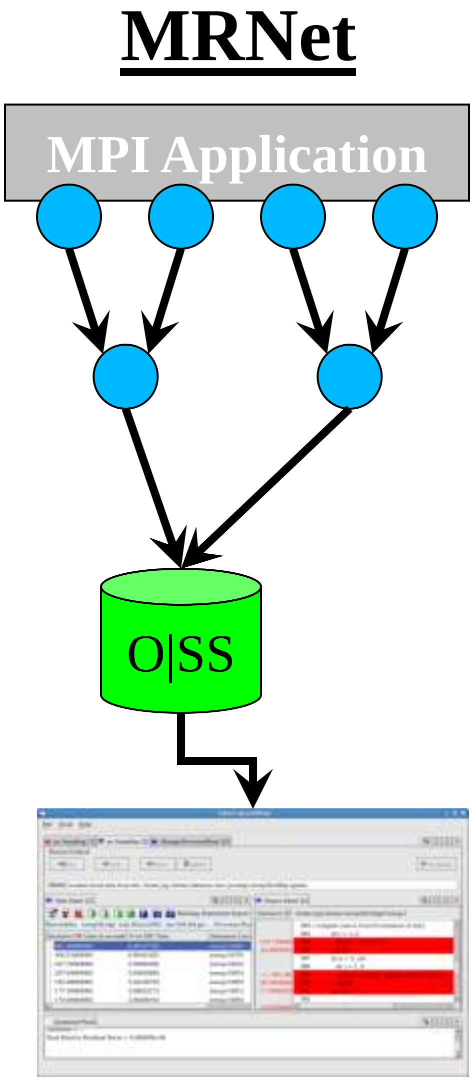
- **Important runtime environment variables**
  - OPENSS_PREFIX (install directory path)
  - OPENSS_PLUGIN_PATH
    - Path to directory where plugins are stored
  - OPENSS_MPI_IMPLEMENTATION (if multiple)
    - If Open|SpeedShop was built with multiple MPI implementations, this points openss at the one you are using in your application
    - Also, only required if using the mpi, mpit, or mpiotf experiments
  - LD_LIBRARY_PATH, PATH
    - Linux path variables

Advanced Features Overview

- **Interactive analysis**
  - "Online"/"MRNet" mode
  - Ability to attach
- **Advanced GUI features**
- **Scripting Open|SpeedShop**
  - Using the command line interface
  - Batch options
  - Integrating O|SS into Python
  - Interoperability

Interactive Analysis

- **Dynamic instrumentation**
  - Works on binaries
  - Add/Change instrumentation at runtime
  - Dynamic attach
- **Hierarchical communication**
  - Efficient broadcast of commands
  - Online data reduction
- **Interactive control**
  - Available through GUI and CLI
  - Start/Stop/Adjust data collection



General GUI Features

- **GUI panel management**
  - Peel-off and rearrange any panel
  - Color coded panel groups per experiment
- **Context sensitive menus**
  - Right click at any location
  - Access to different views
  - Activate additional panels
- **Access to source location of events**
  - Double click on stats panel
  - Opens source panel with (optional) statistics

Leaving the GUI

- **Three different options to run without GUI**
  - Equal functionality
  - Can transfer state/results
- **Interactive Command Line Interface**
  - `openss -cli`
- **Batch Interface**
  - `openss -batch < openss_cmd_file`
  - `openss -batch -f <exe> <experiment>`
- **Python Scripting API**
  - `python openss_python_script_file.py`

CLI Language

- **An interactive command Line Interface**
  - gdb/dbx like processing
- **Several interactive commands**
  - Create Experiments
  - Provide Process/Thread Control
  - View Experiment Results
- **Where possible commands execute asynchronously**

http://www.openspeedshop.org/docs/cli_doc/

CLI Command Overview

- Experiment creations
  - expcreate
  - expattach
- Experiment control
  - expgo
  - expwait
  - expdisable
  - expenable
- Experiment storage
  - expsave
  - exprestore
- Result presentation
  - expview
  - opengui
- Misc. commands
  - help
  - list
  - log
  - record
  - playback
  - history
  - quit

CLI Command Examples

- **Simple usage to create, run, view data**
  - **openss –cli (use cli to run experiment: 3 commands)**
    - **expcreate –f "mutatee 2000" pcsamp** (create an experiment with instrumentation added for the particular collector)
    - **expgo** (runs the experiment gathering data into database)
    - **expview** (displays the default view of the performance data)
- **Alternative views of the performance data**
    - **expview –v statements** (see the statements that took the most time)
    - **expview –v linkedobjects** (see time attributed to the libraries in appl.)
    - **expview –v calltrees, fullstack** (see all the call paths in application)
    - **expview –m loadbalance** (see the min, max, average across ranks/threads)
    - **list –v metrics** (display the optional performance data metrics)
    - **expview –m <metric from above>** (view the metric specified)

User-Time Example

Create experiments and load application named “fred”

```
lnx17>openss -cli
openss>>Welcome to OpenSpeedShop 1.9.3.4
openss>>expcreate -f test/executables/
        fred/fred usertime
The new focused experiment identifier is:-x 1
openss>>expgo
Start asynchronous execution of experiment:
-x 1
openss>>Experiment 1 has terminated.
```

Start application

Showing CLI Results

```
openss>>expview
Excl CPU time Inclu CPU time   % of Total Exclusive Function
in seconds.   in seconds.      CPU Time (defining location)
5.2571           5.2571         49.7297  f3 (fred: f3.c,2)
3.3429           3.3429         31.6216  f2 (fred: f2.c,2)
1.9714           1.9714         18.6486  f1 (fred: f1.c,2)
0.0000          10.5429          0.0000  work(fred:work.c,2)
0.0000          10.5714          0.0000  main
                                          (fred: fred.c,5)
```

CLI Batch Scripting (1)

- **Create batch file with CLI commands**
  - Plain text file
  - Example:

```
# Create batch file
echo expcreate -f fred pcsamp >> input.script
echo expgo >> input.script
echo expview pcsamp10 >>input.script

# Run OpenSpeedShop
openss -batch < input.script
```

CLI Batch Scripting (2)

- **Open|SpeedShop batch example results**

```
The new focused experiment identifier is:  -x 1
Start asynchronous execution of experiment:  -x 1

Experiment 1 has terminated.
   CPU Time  Function (defining location)
    24.2700  f3 (mutatee: mutatee.c,24)
    16.0000  f2 (mutatee: mutatee.c,15)
     8.9400  f1 (mutatee: mutatee.c,6)
     0.0200  work (mutatee: mutatee.c,33)
```

Python Scripting

- Open|SpeedShop Python API that executes “same” Interactive/Batch Open|SpeedShop commands
- User can intersperse “normal” Python code with Open|SpeedShop Python API
- Run Open|SpeedShop experiments via the Open|SpeedShop Python API

Summary

- **Multiple non-graphical interfaces**
  - Interactive Command Line
  - Batch scripting
  - Python module
- **Equal functionality**
  - Similar commands in all interfaces
- **Results transferable**
  - E.g., run in Python and view in GUI
  - Possibility to switch GUI ↔ CLI

Summary / Advanced Features

- **Online instrumentation techniques**
  - Scalable data collection
  - Ability to attach to running applications
- **Flexible GUI that can be customized**
- **Several compatible scripting options**
  - Command Line Language
  - Direct batch interface
  - Integration of O|SS into Python
- **GUI and scripting interoperable**
- **Plugin concept to extend Open|SpeedShop**

Conclusions

How to Analyze the Performance of Parallel Codes 101
A Case Study with Open|SpeedShop

Questions vs. Experiments

- **Where do I spend my time?**
  - Flat profiles (pcsamp exp.)
  - Getting inclusive/exclusive timings with callstacks (usertime exp.)
  - Identifying hot callpaths (usertime exp. + HP analysis)
- **How do I analyze cache performance?**
  - Measure memory performance using hardware counters (hwc exp.)
  - Compare to flat profiles (custom comparisons)
  - Compare multiple hardware counters (additional hwc exp.)
- **How do I identify I/O problems?**
  - Study time spent in I/O routines (io exp.)
  - Compare runs under different scenarios (comparisons)
- **How do I find parallel inefficiencies?**
  - Study load balance between tasks (LB view of pcsamp exp.)
  - Study time spent in MPI routines (mpi exp.)
  - Find outliers with cluster analysis (CA view)

O|SS Documentation

- **Current version: 1.9.3.4**
- **Open|SpeedShop User Guide Documentation**
  - http://www.openspeedshop.org/docs/users_guide/
  - /opt/OSS/share/doc/packages/OpenSpeedShop/users_guide
    - Where /opt/OSS is the installation directory
- **Python scripting API Documentation**
  - http://www.openspeedshop.org/docs/pyscripting_doc/
  - /opt/OSS/share/doc/packages/OpenSpeedShop/pyscripting_doc
    - Where /opt/OSS is the installation directory
- **Command Line Interface Documentation**
  - http://www.openspeedshop.org/docs/cli_doc/
  - /opt/OSS/share/doc/packages/OpenSpeedShop/cli_doc
    - Where /opt/OSS is the installation directory
- **Please : provide feedback!**

Availability and Contact

- **Open|SpeedShop website:**
  http://www.openspeedshop.org/
- **Download options:**
  - Package with Install Script
  - Source for tool and base libraries
- **Feedback**
  - Bug tracking available from website
  - Contact information on website
  - Feel free to contact presenters directly